University of Alberta

PROGRAM CLONING -- AN EVOLUTIONARY APPROACH FOR SOLVING SOFTWARE ENGINEERING PROBLEMS

by

Xinwei Chai

A thesis submitted to the Faculty of Graduate studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**

Department of Electrical and Computer Engineering

Edmonton, Alberta

Spring, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96457-4
Our file Notre référence
ISBN: 0-612-96457-4

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

As an evolutionary-based machine-learning method, Genetic Programming (GP) is widely used to solve diverse problems. However, this powerful but still young algorithm has not been tried on program understanding and reproduction for real programs in any general high-level computer language. In fact, such an automatic program understanding and reproduction process is definitely in need not only for software production processes but also for a variety of software engineering activities. This thesis presents such an approach – Program Cloning, which is based on GP and aims to automatically produce general-purpose programs according to an understanding of the target problems. The Program-Cloning method will significantly benefit diverse software engineering topics such as software testing, software quality assurance, safety-critical software designing, software modeling, software reengineering and agile software engineering. Based on our Program-Cloning experiments and the corresponding results, this thesis demonstrates its principle and possible applications.

Acknowledgements

I would like to thank my supervisors Dr. James Miller and Dr. Marek Reformat, for their support and guidance throughout. I'm grateful to their valuable instructions and ideas regarding this research and their continuously and patiently advising on academic writing.

Thanks also go to Dr. Dave Clyburn, who provided great help on my academic writing for this thesis.

Thanks must also go to my fellow graduate students. Thanks to all the members in the STEAM lab: Zhichao Yin, Ping Li, Bengee Lee, H.C. Yeoh, Maggie Xiao, Howard Zhang and Yunbo Zhou for their priceless advice and discussion.

I would like to dedicate this thesis with my love to my parents, my brother and my sister-in-law in Beijing. Their unconditional love and encouragement always accompany me wherever I go, and empower me whatever I do.

Contents

Chapter 1 Introduction **1**

1.1 Genetic Programming (GP) 4

1.2 Program cloning 7

1.3 Thesis contributions 9

1.4 Thesis outline 9

Chapter 2 Experiments in automatic programming for general purposes **10**

2.1 Introduction 10

2.2 Strongly-typed GP 12

2.3 Statistical results under different GP evolution strategies 17

2.4. Fighting with individual bloating 24

2.5. Representation of diverse program structures 27

2.5.1 Trials on the Triangle.c program 27

2.5.2 Trials on the Extract.cpp program 29

2.5.3 Trials on the Journey.cpp program 33

2.6 Conclusions and the following work 35

Chapter 3 Implementation of the program-cloning package **36**

3.1 Architecture 36

3.2 Implementation Features 37

3.2.1 Strong typing 38

3.2.2 Exception handling 43

3.2.3 Input/Output handling 45

3.2.4 Constant 47

3.2.5 Dynamic fitness calculation 50

3.2.6 Issues concerning implementing GP in Java 52

3.2.7 Sub-function 58

3.2.8 Assigning suitable rates for crossover, mutation and reproduction 60

3.2.9 Number of test cases 61

3.2.10 Dynamic mutation probability and other implementation aspects 62

Chapter 4 Experiments with program cloning using program-cloning package 64

4.1 Triangle.c 65

4.2 Extract.cpp 72

4.3 Journey.cpp 77

4.4 NextDate 82

4.5 Commission 84

Chapter 5 Applying program cloning in software engineering 88

5.1 Complexity measurement 89

5.2 Mutant software test 93

5.3 N-Version software design 95

5.4 Test first 97

5.5 Test data evaluation 98

5.6 Automatic test data generation 100

Chapter 6 Conclusions and Future work 106

6.1 Conclusion 106

6.2 Future work 107

Bibliography 110

List of Tables

Table 2.2.1 Key features 16

Table 2.3.1 Attributes and parameters for each trial 19

Table 2.5.2.1. Parameters and settings for trials of *Extract.cpp* 32

Table 3.2.6.1 Main node set defined in PCP 54

Table 3.2.8.1 Genetic-operation proportion 60

Table 4.1.1 Constraints of basic function-types 68

Table 4.2.1 Different parameters for trials with *Extract.cpp* 75

Table 4.3.1 Decision table of *Journey.cpp* 79

Table 4.4.1. Designing fitness cases 83

List of Figures

Figure 1.1.1 GP's principle 4

Figure 1.1.2 Crossover 7

Figure 1.1.3 Mutation 7

Figure 2.2.1 Code segment of *Journey.cpp* 14

Figure 2.3.1 Code of *Triangle.c* 18

Figure 2.4.1 Performance with double tournament 26

Figure 2.4.2 Performance with dynamic parsimony pressure 26

Figure 2.5.1.1 Perfect solution expressed in C 29

Figure 2.5.2.1 Code of *Extract.cpp* 30

Figure 2.5.3.1 Code fragment of *Journey.cpp* 33

Figure 2.5.3.2 Cloning program translated in C language 35

Figure 3.1.1 Architecture of PCP 36

Figure 3.2.1.1. Four invalid individuals 39

Figure 3.2.4.1. Parse tree with constant terminals 49

Figure 3.2.5.1 Pseudo code for individual fitness calculating 51

Figure 3.2.10.1. Diversity of population 63

Figure 4.1.1 A solution tree for triangle 65

Figure 4.1.2 Comparing best programs from GP trials with 60 and 30 cases 69

Figure 4.2.1 Simplified solution in trials with *Extract.cpp* 77

Figure 4.3.1 The Best solution 1 in pseudo code 80

Figure 4.3.2 The Best solution 2 in pseudo code 81

Figure 4.5.1 Adopting Array 85

Figure 5.5.1 Test data evaluation 99

Figure 5.6.1 Hand-written test data generator for *Triangle.c* 104

Figure 5.6.2 Automatic test data generation 105

Chapter 1

Introduction

Software engineering arose out of the software crisis in the late 60s, however, despite many innovations, software development is still in a state of crisis. Studies from Standish Group* have shown that 31.1% of projects will be canceled before they ever get completed. Further results indicate that 52.7% of projects will cost 189% of their original estimates. According to the Standish Group, the United State spends more than $250 billion each year on IT application development of approximately 175,000 projects. Of these 31% are cancelled, 53% are changed and 16% successful.

The root cause of software crisis is the very character of software – different from normal products, software is the direct product of human intelligence. As described by Brooks [Broo86], complexity, conformity, changeability and invisibility construct the essential difficulties for software development. In addition, accidental difficulties exist, relating to the production of software. Software engineers and project managers have tried and are trying different software engineering process, methods and tools in order to economically obtain reliable and efficient software. Current directions for software engineering include agile software engineering, aspect programming and the unified software development process. However, while the wide applicability of stock hardware has driven demand up and manufacturing cost per unit dramatically down, the same trends have led to increasing demand for ever more complex software for an astounding variety of uses [Gumn96]. "Large" programs now can have 1,000,000 to 5,000,000 lines, and some commercial software, such as Microsoft Word or Microsoft Excel are moving into the 10,000,000+ range [Coll03]. In contrast, due to the very character of software, the vast majority of computer code is still "handcrafted from raw

* The Standish Group. CHOAS Chronicles II, The Standish Group International Inc., 2001

programming languages by artisans using techniques that are neither measurable nor repeatable consistently", and "software seems like malleable stuff, most programs are actually intricate plexuses of brittle logic through which data of only the right kind may pass" [Gibb94]. As a result, success in software development still depends most upon the quality of the people involved. Although the art of programming has been continually refinement for more than half a century, software development has reached a stage that the current development processes and models struggle to cope with the complexity of modern applications.

In Brooks' [Broo86], automatic programming is mentioned as one technical development that are most often advanced as a potential "silver bullet". However, automatic programming is only commonly found in rule-based problems. When handling high-level languages, current automatic programming focuses on glamour but not semantic content and struggles to generalize to industrial systems. Still, automatic or semiautomatic programming or problem learning provides a potential solution to software crisis, and eventually, the automatic programming technologies will become mature enough to provide a silver bullet for software development.

To contribute to the maturity of automatic programming, our research tried to work out an approach to automatically comprehend a problem and find its solution programs in high-level languages. Diverse heuristic machine learning methods or algorithms may contribute to the automatic programming, for example, Genetic Programming (GP), Simulated Annealing and Grammatical Evolution [Onry01]. Genetic programming is a young and powerful machine-learning algorithm. Using hierarchical structures that are dynamically shaped, GP can express many logic-included structures including conditions, iterations and combined data types. As an evolutionary algorithm, GP has the capability to solve problems which are too complex to be well understood, which is accord with the demand for problem understanding – the most difficult part in software development. Our research is based on GP. As an initial try to apply GP on automatic programming using standard computer languages for arbitrary problems, our experiments focus on automatic comprehension and solution production for a series of simple problems. In addition, our automatic problem learning/programming approach aims to solve a series of concrete problems in software engineering

areas including software testing, software complexity measurement and test-driven software development.

Based on the Genetic Algorithm (GA), Genetic Programming (GP) adopts more complex individual structures for its genetic evolution process and is widely applied as a machine-learning algorithm. Since Koza introduced GP in 1992 [Koza92], it has been adopted in a broad variety of problems, such as image processing, electrical circuit designing, robot controlling and system modeling and designing in economics and biology domains. However, since GP is still a comparatively new field, its potential uses in a large number of other domains in which heuristic machine-learning processes are required have not been investigated to date. In this thesis, a new method based on GP, program cloning, is proposed and explained by our trials. In addition, program cloning's most promising applications in software engineering are proposed and analyzed.

Unlike GP's typical applications, such as symbolic regression, artificial ant and 11-multiplexer [Koza92], program-cloning aims to automatically create "general-purpose programs" based on an understanding of the original program's solution for a certain problem or of the problem's specification. Because of its "general purpose" target, the GP process of program-cloning puts greater emphasis on the language elements used to compose individual solutions, not the specific problem details. In conventional GP programs, individual programs are represented by symbolic or operator-like elements focused on formula or rule-like solutions. For example, one of GP's typical application the simple symbolic regression is to search the mathematic expression $y=x^4+x^3+x2+x$; and to solve this problem, limited arithmetic operations like +, -, * and / are generally adopted as GP's function types. In contrast, program cloning uses more complex and flexible language elements and can simulate more complex logic in a program with the final target of competing with hand-written programs. For example, basic programming elements such as arithmetic operations, logic operations, comparing operation and program flow control structures are included in function set to solve problems with whatever purpose like information abstraction, data processing or commercial computing (see chapter 4).

In this chapter, in order to introduce the profile of program cloning step by step, we will cover the

following topics in the subsequent four sections:

1) Genetic Programming,

2) Program cloning,

3) Thesis contributions, and

4) Thesis outline

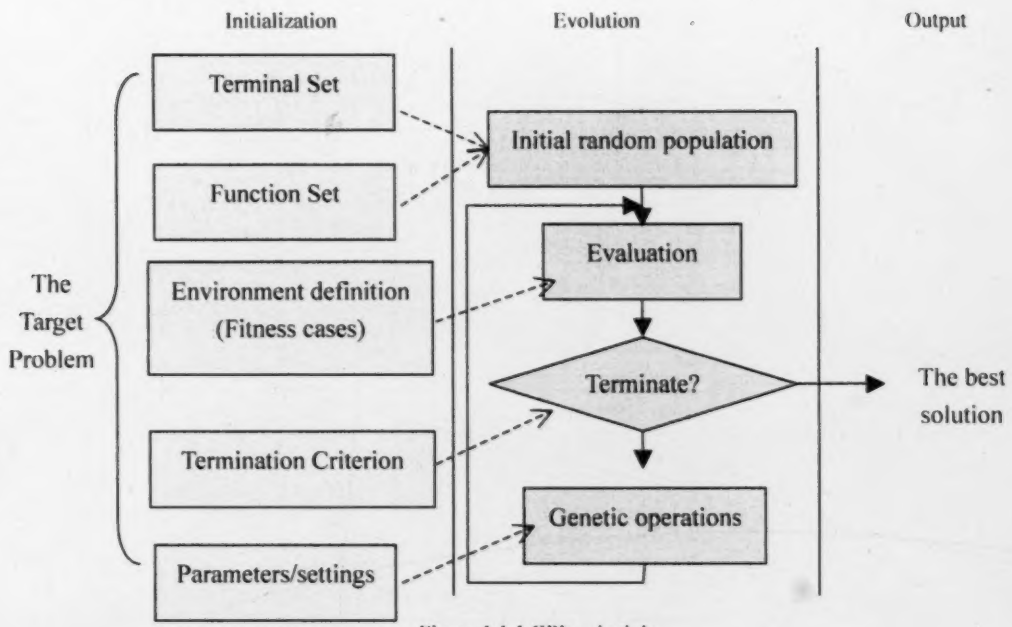


Figure 1.1.1 GP's principle

1.1 Genetic Programming (GP)

Genetic programming (GP) is an extension of the Genetic Algorithm (GA). As important evolutionary algorithms, GA and GP are widely applied in order to solve problems whose solution spaces are too complex to be understood or efficiently explored by traditional methods. The basic idea of GA originates from Darwin's natural selection theory. In nature, creatures on earth have been evolving generation by generation according to the survival of the fittest law; in science, a solution in a certain problem-specified environment can also be evolved generation by generation for improving solutions to a target problem. GP extends GA by adopting hierarchical individual structure, whose content and shape are both dynamically constructed, instead of GA's fix-length strings individual structure, whose shape are statically determined. For example, to solve the eight-queens problem, GA's individuals can

adopt a numeric array which has fixed 8 elements; in contract, GP's individuals generally adopt a tree-based structure with diverse shapes like the trees in Figure 1.1.2. This character enables GP to solve more complex problems and to be an ideal machine learning method in artificial intelligence. GP's main process can be illustrated by Figure 1.1.1.

According to the specific problem, initialization work before GP evolution is required. The initialization includes determining elements in the terminal and function sets, defining the evolving environment, which is generally a data set of fitness-evaluation cases, determining termination criterion and configuring parameters and settings within GP algorithm. In the GP's evolution, an initial population, which includes a predefined number of solution individuals, is established according to certain individual construction rules, and then, the individuals are evaluated under the defined environment. Then, the termination criterion is examined to determine whether to terminate the GP evolving and output the best solution or to perform genetic operations to create the next generation and then enter the next GP evolving loop. To understand the GP algorithm, some key-concepts must be clarified, and we list and explain them as the following.

Function and *terminal sets:* Terminal and function are two kinds of basic components for constructing GP's solution individuals, these components are genes, and in this thesis they are also mentioned as primary elements. Generally, the terminals represent interface variables for the problem and basic operations and necessary constants that can be useful to express a solution. The functions represent parameter-needed operations, such as arithmetic operations, mathematical functions, logic operations, conditional operators, iteration control structure, flow control operations and any other domain-specific function that can be used for the target problem. To solve the target problem, the terminal set and the function set must be sufficient enough; to make sure that every solution individual created by mechanically combining primary elements is valid to GP evolving process, the terminal set and the function set must be closured.

Individual, population and generation: In GP, one hierarchical structure (generally tree-based structure) that is composed by primary elements is a potential solution to the target problem and is an individual in GP's evolving process. To carry out selection based on the survival of the fittest, a

population, composed of a certain number of individuals, is used in GP's evolving process. The individuals of a population keep changing along with the evolving iteration, and each loop is one generation. The population in the first generation, the initial population, is composed of individuals which are created by combining elements randomly selected from the function set and the terminal set.

Problem environment: For natural selection, the evolution environment is the surface of earth; for GP's selection, the evolution environment is the outside behavior of the target problem. In GP, a problem's behavior can be reflected by data sets or by a feedback source [Kush02], and all trials in this thesis adopt the former method. The cases in the data sets are called fitness cases with respect to GP algorithm or test cases with respect to the programs being cloned.

Fitness measure: Fitness measure is a quantitative way to express how well an individual to fit the problem environment or to solve the problem, and in this thesis, fitness measure is also mention as fitness function or object function. Fitness measuring is based on the problem environment definition, and like natural selection, selecting individuals to enter or partly enter the next generation is based on the fitness values of individuals, and fitness thus acts as the driving force in GP's evolution. Koza [Koza92] has introduced four fitness measures: raw fitness, standardized fitness, adjusted fitness and normalized fitness, and our trials in this thesis adopted the first three.

Genetic Operations: To achieve steady improvement of the population's fitness, individuals from the second generation are created by carrying out a series of operations on selected individuals of the previous generation. These operations are called genetic operations, and the most frequently used genetic operations are crossover, reproduction and mutation.

1) Crossover Operation: As shown in Figure 1.1.2, in a crossover operation, two parent-individuals exchange sub-trees rooted at the randomly selected nodes to form two child-individuals.
2) Reproduction: Reproduction is asexual because it is carried out on one parent, and it simply duplicates the parent individual to create a new one, the offspring.
3) Mutation: As shown in Figure 1.1.3, mutation is also asexual. In the mutation operation, one individual is selected based on its fitness, from this individual's tree structure, a mutation point is randomly chosen, and then, the original sub-tree rooted at this point is substituted by a new one,

which is constructed by combining randomly selected primary elements.

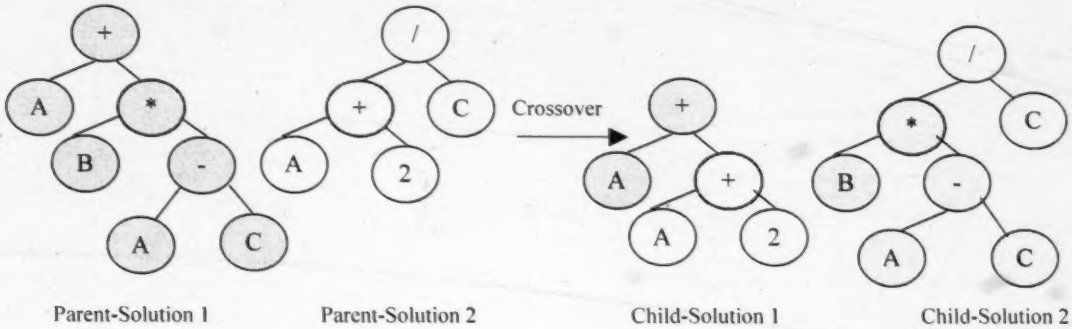


Figure 1.1.2 Crossover between parent-solution 1 and 2, where the red-circled node is selected to carry out crossover

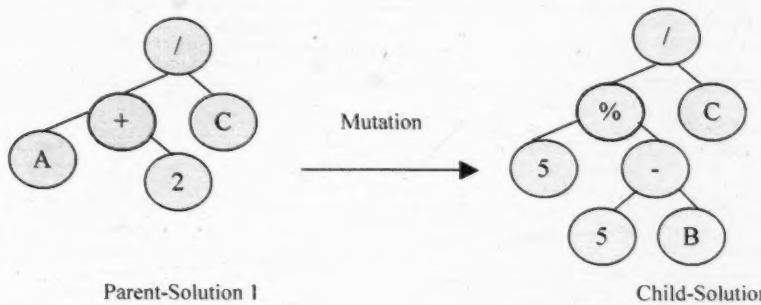


Figure 1.1.3 Mutation upon parent-solution 1, where the red-circled node is selected to carry out mutation

1.2 Program cloning

Operating on fixed-length character strings, GA can resolve only certain problems that can be solved by simple and static solutions; to overcome this defect, GP was devised based on GA. GP represents solutions by hierarchical computer programs [Koza92], which enables it to provide more complex, flexible and logical solutions than GA. For example, conditional and iteration operations are usually used in GP to solve certain problems. Based on Koza's claim [Koza92] that "for problems that can be viewed as requiring discovery of a computer program that produces some desired outputs for particular inputs, ... GP provides a way to search for this fittest individual computer program", we believe that GP provides a way to understand a problem and its solutions as a black box and to work out a solution or substitute solutions as a white box for that problem. This belief is one of program cloning's principles. Program Cloning is the production of a system from its external interactions [Mirc04]. Program cloning's another basis is that instead of being limited to artificial intelligence,

symbolic processing, and machine learning domains, GP can be extended to apply on any problem as long as its specification can be fully reflected by input/output sets and its solutions can be expressed in a program style.

The program-cloning concept arose from our attempt to apply GA to automatic software-test-data generation. Aiming to reach a certain testing coverage, we kept modifying the test cases in order to meet predications at each branch of the software under test (SUT). The intermediate variables, which often appear in branch predications, have to be substituted by input variables because input variables are the only values that we can adjust in the test cases. Hence, we must find a way to enable the test-data-generation program to understand how to express or deduce the intermediate variables from the input variables, and this is where the GP is introduced. This procedure can be understood as a series of steps to understand a program and then to express this understanding, and the program being understood may serve for any problem and may be in any language.

The main idea behind program cloning can be summarized as follows: 1) it is a GP application for automatically creating programs, 2) these programs have general purposes, 3) these programs are similar to hand-written programs using a general high-level computer language, 4) through the input/output set, it tries to understand the problem specification or the original program, and 5) the new program reflects the understanding achieved in 4).

A recently developed topic, grammatical evolution (GE), [Onei01] [Onry01] also aims to generate programs in an arbitrary language, and its key approach is to represent individuals in variable-length binary strings, through grammar definition using Backus–Naur forms (BNF) and genotype-to-phenotype mapping process to produce and evolve programs. Compared with program cloning, GE, however, focuses only on producing the final programs in arbitrary languages through explaining BNF, not general-purpose programs. In contrast, program cloning puts emphasis on the practices of creating general purpose programs and takes it for granted that post-processing manipulation can easily translate solutions into any language.

1.3 Thesis contributions

With respect to the program-cloning problem, there are three contributions in this thesis. 1) Adopting ECJ, which is a general-purpose genetic programming package [Luke02], we experimented with a series of trials utilizing GP to simulate programs with diverse structure characters. 2) Based on the experiments involved in 1), we designed and implemented a trial version GP package, which focuses particularly on the program-cloning problem, and using this package, we carried out more trials to clone programs with different levels of complexity, according to the original program of a problem or the problem specification. Typical samples are selected and described in this thesis. 3) Based on our trials, we propose six possible applications in software engineering domain that can benefit from the program-cloning technique.

1.4 Thesis outline

The content of this thesis is organized as following. Chapter 2 discusses program cloning experiments based on a general-purpose GP package; Chapter 3 described the implementation of the trial version of a program-cloning specific GP package; Chapter 4 discusses typical samples of program cloning experiments based on our program-cloning specific GP package; Chapter 5 proposes hopeful application domains for program cloning; and Chapter 6 offers conclusions and suggestions for future work.

Chapter 2

Experiments in automatic programming for general purposes

2.1 Introduction

Genetic programming (GP) is an important approach in automatic programming. Its typical application includes diverse symbolic expression problems, numerical or logical expression seeking and game-solution problems. However, its real attraction is the potentiality for automatically implementing complex programs, which are normally written by programmers in general languages like C and Java. This chapter describes our initial attempts to automatically create programs to implement arbitrary problems or clone general-purpose programs. The experiment described in this chapter is based on Luke's general-purpose GP package, ECJ version 10, [Luke02]; our attempt in this chapter is called automatically programming for general purposes (APGP).

To simulate a real program, the key for the GP algorithm is the individual's structure, which must be applicable to a program with any arbitrary purpose. To compose such an individual structure, both the node set and constructing rules are quite different from those in a traditional GP application.

First, the nodes should provide the flexibility and efficiency to construct an arbitrary program, which mimics the commands and control instructions found in general languages. In a typical GP problem, the terminals and functions in the node set are correspondingly simple and problem-specific. For different GP applications, their node sets are generally totally different. For example, the node set for the ant problem [Koza92] generally includes "if-Food-Ahead", "left", "right" and "move"; and for the symbolic regression problem [Koza92] includes "add", "sub", "mul", "div", "exp" and "log". However, in our problem of automatic programming for general purposes (APGP), the node set of GP must include functions such as "if-then-else", "loop", "assignment" and "procedure definition" as well as some optional function packages, which mimic libraries in a programming language. For example,

when cloning programs on numerical problems, the APGP should include the package of "math".

Second, for the APGP problem, the individual structure of GP is much more complex and flexible than that for traditional GP problems. For example, it is very common for an arbitrary program to declare a temporary variable, assign a value to it and retrieve the value from this temporary variable. So, special data structures and operations should be designed to clone these operations. At the same time, in order to make the GP searching more sufficient, the construction rules should be restrictive enough to guarantee that all of the individuals are valid. The strongly typed technique is an efficient approach for GP to handle complex individual structures, and simultaneously preserve the closure attribute of the node set. In the APGP problem, the application of strongly typed technique brings new content. For instance, for a general application of strongly typed GP, when matching a child node to a parent node in an individual, we need only consider their data-types; In the APGP, only checking the nodes' data-types is often insufficient, e.g., to a parent function of "integer assignment", its first child (the left hand side of the operation) should be not only an integer but also a variable, and a constant integer can not be matched here.

Besides the individual structure, there are many other factors that affect the GP's efficiency, e.g. individual-bloat control approaches, random number generators, selection methods and GP parameter settings. Our experiments focused mainly on the following four aspects:

1) Strongly-typed GP,

2) Comparison of GP's behavior under different operations, selection methods, variable domains and bloating refraining approaches,

3) Experiments with individual bloating control and

4) Representation of diverse program structures.

All our experiments were based on a GP package, ECJ [Luke02]. This package was developed by Sean Luke in Java, and comprehensively implements GP system. It provides diverse facilities such as Mersenne Twister Fast-random-number generator, strongly typed, Ephemeral Random Constants and Automatically-Defined Functions [Koza92]. ECJ was designed to be flexible and modifiable by using hierarchical parameter files, which defines tens to hundreds of parameters to establish the whole

algorithm strategy. As a general purpose GP package, ECJ is very efficient for most of the typical GP problems.

However, when applying the ECJ package to the APGP problem, we had to rewrite a number of the basic classes like the individual builder (JoGrowBuilder), the individual class (JoGPIndividual) and the crossover operation class (JoCrossoverPipeline). Unfortunately, there are still several un-applicable points in ECJ to the APGP problem, which are difficult to improve and thus impose a limitation on our trials, for example, the strongly typed construction for an individual. On the other hand, the whole mechanism of ECJ, which was delicately designed to achieve generality for diverse GP problems, is unnecessary in our research. In fact, a program-cloning specific implementation of GP is definitely helpful in order to carry out an extensive research on this problem, and the relative experiments and studies will be described in the next chapter, and the experiments using ECJ provide basis for further work.

In the following sections, we will depict the four experiment aspects respectively; finally, in the conclusions section, we will summarize the experiments in this chapter and point out the direction for further work.

2.2 Strongly-typed GP

In the standard GP, elements in the function set and the terminal set generally have no data type or single data type. For example, in the typical GP application, simple symbolic regression problem, whose target function is $x^4+x^3+x^2+x$, every element in GP adopts float data type. In order to carry out genetic operations on the individuals, the elements that are used to construct individuals must satisfy the requirements of closure and sufficiency. Although for each target problem, a particular GP application is designed, there is no way to guarantee that its individuals satisfy some complex constraints of the target problem. Consequently, the solution space is unnecessary huge and the large number of invalid individuals depress the success rate badly.

Then, it is natural for people to resort to strongly typed genetic programming (STGP). Strongly typed genetic programming is an enhanced version of genetic programming [Mont95] which enforces

data type constraints and whose use of generic functions and generic data types makes it more powerful than other approaches to type constraint enforcement [Mont95]. When applying GP to solve a certain problem, without losing generality, the more restrictions added to the individual, the smaller the solution space is and the greater the probability of locating the target solution becomes. Before the proposal of STGP, Koza [Koza92] has defined constrained syntactic structures, by which he defined syntaxes by directly specifying what kind of children each non-terminal can possess. The basic STGP does this indirectly by specifying the data types of each child and the return types of each node [Koza92]. Although loses a certain degree of flexibility, the data type concept provides basis for the techniques of generic functions and generic data types. The generic functions and generic data types aim to provide GP with the possibility of not only handling problems such as the symbolic manipulation of vectors and matrices but also the ability to create large and complex programs rather than just small and simple programs [Mont95].

In our experiments in applying GP to the APGP problem, we found that in order to represent an arbitrary program, both Koza's constrained syntactic structures [Koza92] and Montana's STGP with data types specifying [Mont95] are helpful. Data type specifying makes it possible to handle multiple data type conditions and clone the data typing operations in general programming languages. However, under certain conditions, it is necessary to define more detailed constraints, i.e. what kind of children each non-terminal can have. For example, for the non-terminal function "integer assignment", its first child should be not only integer but also a variable, and a constant integer does not match here. Montana's generic function, similar to the template concept in C++ language, is a function that can take a variety of different argument types and, in general, return values of a variety of different types [Mont95]. The generic data type, similar to the abstract class in object oriented language, is not a true data type but rather a set of possible data types [Mont95]. The generic function and the generic data type can be used to clone special usages in a programming language. However, Montana's other three proposals, void data type, local variables and run time error [Mont95], are obviously very applicable in the APGP problem.

In an arbitrary program, there are many control structures that cannot be represented in normal

ways. Under these conditions, the easiest method is to add the particular constraints directly into the rules of individual representation. For example, for the "if-then-else" function node, its first child is functional only if it is a logic expression; hence, the first child should be a non-terminal node; and consequently, the "if-then-else" should not occur in the last second layer in a tree-based individual. In a GP implementation that focuses only on the APGP problem, designing a mechanism for adding, selecting and modifying those specific constraints is not difficult. But for a general GP package like ECJ, adding a complex constraint is difficult and often depresses the efficiency of the original package.

The following is one of our experiments, which clones a code segment in an arbitrary program, *Journey.cpp*. The background of *Journey.cpp* problem is the Collatz Problem or the 3x+1 Problem [Vard91], which is defined as: the series a_n (n=0, 1, 2, ...), which equals to a_{n-1} / 2 for an even a_{n-1} and 3 * a_{n-1}+1 for an odd a_{n-1}, always converges to 1 for any positive value of a_0. For example, when a_0 equals 3, the series is 3, 10, 5, 16, 8, 4, 2, 1; when a_0 equals 5, the series is 5, 16, 8, 4, 2, 1. The *Journey.cpp* program calculates the series for each number (a_0) in an integer set, and locates the maximum value among the peak-values of the series.

```
int n = current; //current is an init value
int peak=n;
if(n % 2){
    n = 3 * n + 1;
    if(n < 1){
        cerr << current << " Integer multiplication failure\n";
        exit(2);
    }
    if (n > peak)
        peak = n;
} else
    n = n / 2;
```

Figure 2.2.1 Code segment of *Journey.cpp*

Our trials focus on the code fragment that calculates the series (a_n), and modifies *peak* if *peak* is smaller than a_n. The code segment is shown by Figure 2.2.1. This program includes three if-then-else structures, two input variables, *n* and *peak*, and two output variables, *n* and *peak*.

Table 2.2.1 summarizes the key features in this trial.

Objective:	To search an arbitrary program that implements the same function as the target code segment.
Data Types:	Integer, Boolean and float [1].
Terminal Set:	Variable n, variable peak and a random constant by range (-10,10) in integer.
Function Set [2]:	+, -, *, /, %, =, >, <, Assignment, If, Sequence (to represent sequentially execution of commands)
Node constraints:	Individuals return float values [1]; variable n and variable peak adopt integer values; arithmetic operations, +, -, *, /, % adopt 2 integer children and return integers; comparing operations, =, >, < adopt 2 integer children and return Booleans; Assignment [3] adopt 2 integer children with the first child representing a variable and return floats; Sequence adopts 2 float children and returns floats; and If adopts 3 children with data types Boolean, float and float respectively and returns floats.
Fitness Calculation:	20 test cases are designed according to the code segment; the evaluation process for an individual clone is: given input and output values by 20 test cases, the values of a cloning program are matched with the output, one unit is scored per successful output; perfect score is 40 units; standard fitness = raw fitness = the number of un-matching cases; and adjust fitness = 1/(1+standard fitness).
Hits:	The number of matching cases.
Parameters and settings [3]:	Population number = 4000, Maximum generation = 501, Cross-Over Probability = 0.7, Mutation Probability = 0.25, Reproduction Probability = 0.05, Tournament Selection
Success	Adjust fitness = 1.0 or Hits = 40

Predicate:	
Specific rules of construction:	Since the function nodes If and Sequence must not have children of terminal node, they should not occur in the second last layer in a tree-based structure.

Table 2.2.1 Key features

[1] All of the float data types here should be void types, and there are two reasons that we use float here: firstly, it is necessary to clone the "*exist (2)*" command in the code segment, instead of Mantana's run-time-error due to the ECJ package, we utilize a data type to represent whether the execution is successful executed; Secondly, although a Boolean (or integer) type seems more reasonable instead of float type, the Boolean (integer) type has been used for variables and the STGP in ECJ package does not distinguish nodes with same data type, e.g., a "Sequence" node with two Boolean children may be given a child node, ">" or "<". Thus, in order to simplify the trial, we simply utilized the float data type.

[2] Function set includes two classes of nodes: the first class is for arbitrary program representation such as Assignment, Sequence and If, and the second class is for numerical operations such as +, -, *, /, %, =, >, <. Although all these operations are very basic for a programming language, a little larger node set may depress the GP's efficiency significantly. Hence, optional function set packages, each of which is assigned a certain probability of being selected, may compensate for GP's efficiency in the APGP problem.

[3] Since APGP is not a typical GP application, standard experiential parameters are not quite available for it. In this chapter, the parameters and settings listed are experiential values derived from our trials. However, the parameters or settings given here are not the only applicable ones.

The above three annotations are applicable to all of the trials in this chapter.

A typical successful individual is shown below, where "Assignment" is represent by ":=" and "Sequence" by "->": *(-> (if (> n (+ (% n n) (* (% (% n 2) peak) peak))) (:= n (/ n 2)) (:= n (+ 1 (* 3 n)))) (if (> peak n) (:= peak peak) (:= peak n)))*

Although some meaningless logic exists in this individual, most of the functions in the original code segment are implemented and most of the operations in this individual are reasonable.

For the 53 trials we executed, two solutions with perfect scores are achieved: one with 423

generations and the other with 327 generations. The low successful rate is due to the unnecessary entangling between the two variables. Trials on a similar code segment that contains two independent variables achieve significantly higher successful rate. This issue is discussed in depth in Section 2.4. Moreover, most of the trials reach a final score greater than 36 out of 40 within 501 generations. We believe that an increased rate of perfect scores (i.e. perfect cloning) can be achieved by increasing the maximum number of generations.

2.3 Statistical results under different GP evolution strategies

As part of our experiments with GP, we compare the GP's results under the following different conditions:

1) Three genetic operations, crossover, mutation and reproduction against two genetic operations, crossover and reproduction,
2) Greedily over-selecting versus tournament selecting,
3) A problem solution domain of [-10,10] against a problem solution domain of [-100,100] and
4) Diverse bloating refraining approaches.

All of the trials are carried out on cloning the same function of *Triangle.c*, the most widely used example in software testing literature [Jorg02]. The function is defined in Figure 2.3.1.

```
int triang(int i, int j,int k){
    int tri=0;
    if ( (i<=0) || (j<=0) || (k<=0) )
        return 4;
    if(i==j)
        tri +=1;
    if(i==k)
        tri +=2;
    if(j==k)
        tri +=3;
```

```
if(tri==0){
    if( (i+j<=k) || (j+k <=i) || ((i+k) <=j) )
        tri=4;
    else
        tri = 1;
    return tri;
}
if ( tri > 3)
    tri = 3;
else if ( (tri == 1) && ((i+j) >k) )
    tri =2;
else if ( (tri == 2) && ((i+k) >j) )
    tri =2;
else if ( (tri == 3) && ((j+k) >i) )
    tri =2;
else
    tri = 4;
return tri;
}
```

Figure 2.3.1 Code of *Triangle.c*

Except further specifications, the attributes and parameters for each trial are shown in Table 2.3.1.

Objective:	Search for a program that implements the same function as the target code segment.
Data Type Specifying:	Integer and Boolean
Terminal Set:	Variable i, Variable j, Variable k and a random constant integer with the same range as i, j, k
Function Set:	+, -, *, =, >, <, Assignment, If

Node constraints:	Return type: integer; i, j, k: integer +, -, *: 2 integer children, return integer; =, >, <: 2 integer children, return Boolean; If: 3 children of data type Boolean, integer and integer, return integer.
Fitness Calculation:	18 test cases are designed according to the target function; the evaluation process for an individual clone is: given inputs from the test cases; score one unit per successful output from the test cases; perfect score = 18 units; standard fitness = raw fitness = the number of un-matching cases; adjust fitness = 1/(1+standard fitness)
Hits:	The number of matching cases.
Parameters and setting:	Population number = 1500, Maximum generation = 251, Cross-Over Probability = 0.9, Mutation Probability = 0.0, Reproduction Probability = 0.1, Tournament Selection
Success Predicate:	Adjust fitness = 1.0 or Hits = 18
Rules of construction:	Since the function nodes, If, must not have children of terminal node, it should not occur in the last second lay within a tree-based structure.

Table 2.3.1 Attributes and parameters for each trial

In the following paragraphs, we will present our experiments under four conditions respectively.

1) Three genetic operations, crossover, mutation and reproduction against two genetic operations, crossover and reproduction: As Koza illustrated, the genetic operations need not include mutation because the crossover operation has a side effect similar to mutation. However, this point of view is not accepted by all of the researchers and many GP applications insist on the mutation operation in the evolution process. Hence an unanswered research questions is: what is the effect of mutation in the APGP domain?

   The crossover utilized here is based on Koza's "Sub-tree Crossover" [Koza92] along with strongly typed consideration. Firstly, two individuals are selected, then a single tree (chromosome)

for each individual is chosen. Then, a random node is selected in each tree such that the two nodes have the same return type. Finally, if by swapping the sub-trees at these nodes, the two new trees will not violate the maximum depth constraints, the swapping is performed and the two new trees are created; otherwise, repeating searching for random nodes is carried out.

The mutation utilized here is the strongly typed version of the Koza's "Point Mutation" [Koza92]. Additionally, we have utilized the tree depth restrictions on the mutation operator, if the tree gets deeper than the maximum tree depth, the new sub-tree is rejected and another tree is created and evaluated.

The reproduction operation is very simple, it makes a copy of the individuals, which are selected according to a certain selection method, like fitness-proportionate selection or tournament selection. If an individual has already been cloned, it will not be cloned again [Luke02].

In the trials with only crossover and reproduction operations, the probability of crossover and reproduction are set as 0.9 and 0.1 respectively. The result is:

- Number of independent trials: 15,
- Number of success trials: 6,
- Successful rate: 40%

In the trials with crossover, mutation and reproduction operations, the probability of crossover, mutation and reproduction are set as 0.9, 0.1 and 0.1 respectively. The result is:

- Number of independent trials: 23,
- Number of success trials: 9,
- Successful rate: 39.1%

According to the results, no obvious effect by mutation operation exists.

2) Greedily over-selecting versus tournament selecting: Greedily over-selecting is generally based on fitness-proportionate selection. It is recommended by Koza [Koza92] for problems where the population size is larger than 1000. Although tournament selection generally seems to be favoured by GP researchers, the unresolved question remains: can it match with Greedily

over-selecting in case of large population? This trial is significant because for the APGP problem, the GP evolution generally need very large populations.

In the first set of trials for greedily over-selecting, the population size is 1500; greedily over-selecting is carried out on the crossover and reproduction operations, each of which has an operation chance of 0.9 and 0.1; and 16% of the population with superior fitness have a 80% chance of being selected. The result is:

- Number of independent trials: 12,
- Number of success trials: 2,
- Successful rate: 16.7%

In the second set of trials for greedily over-selecting, the population size is 4000; greedily over-selecting is carried out on the crossover, mutation and reproduction operations, each of which has an operation chance of 0.7, 0.25 and 0.05; 31 test cases were used to calculate the fitness; the maximum number of generations is 1501; and 8% of the population with superior fitness have a 80% chance of being selected. The result is:

- Number of independent trials: 25,
- Number of success trials: 0,
- Successful rate: 0%

In the first set of trials for tournament selection, the population size is 1500; the tournament selection is carried out on the crossover and reproduction operations, each of which has an operation chance of 0.9 and 0.1; and the tournament size is 7. The result is:

- Number of independent trials: 15,
- Number of success trials: 6,
- Successful rate: 40%

In the second set of trials for tournament selection, the population size is 4000; the tournament selection is carried out on the crossover, mutation and reproduction operations, each of which has an operation chance of 0.7, 0.25 and 0.05; 31 test cases were used to calculate the fitness; and the tournament size is 7. The result is:

- Number of independent trials: 25,
- Number of success trials: 3,
- Successful rate: 12%

It is obvious that tournament selection behaves much better than the greedily over-selecting.

3) A problem solution domain of [-10,10] against a problem solution domain of [-100,100): With the expanding of the range of the solution domain, the solution space expands and the successful rate of GP declines. However, as long as the decline in the efficiency is not too significant, the application of GP should still be an acceptable option.

In our experiments, variables and the random integer constant have the same ranges of values, [-100,100] or [-10,10), the population size is always 2500. The result is:

Smaller range condition [-10,10]:

- Number of independent trials: 24,
- Number of success trials: 18,
- Successful rate: 75%

Larger range condition [-100,100]:

- Number of independent trials: 116,
- Number of success trials: 8,
- Successful rate: 6.9%

Although the successful rate drops significantly, in the trials with the range of values of [-100, 100), we noticed that 86 out of 116 trials meet more than 15 out of 18 test cases. Thus by increasing the population or allowing more generation iteration, we believe that a significantly greater high successful rate can be achieved. Hence, the GP behavior is still considered to be acceptable.

4) Diverse bloating refraining approaches: One of the foremost challenges to scaling genetic programming is bloat, which is the tendency of GP's individuals to grow in size along with the evolution process. Among the diverse bloat control techniques, we experimented with parsimony pressure method [Koza92][Soul98], strength pareto evolutionary algorithm (SPEA2) with the

second object as the size of individuals [Blbr01] and Sean Luke's nonparametric parsimony pressure method [Lupa02]. Listed here is part of the trial results, and the detailed description is given in Section 2.4.

Parsimony pressure method with the parametric coefficient 0.001: 31 test cases with the range [-100,100] are utilized in the fitness calculation; maximum number of generations is 1501; population size is 4000; selection method is tournament selection; and three operations, mutation and crossover and reproduction, are adopted with the probability of 0.7, 0.25 and 0.05 respectively; maximum initial individual depth is 4; maximum individual depth after crossover is 8. The result is:

- Number of independent trials: 24,
- Number of success trials: 0,
- Successful rate: 0%
- Average value of the best individuals: 21.4

Parsimony pressure method with the parametric coefficient 0.01: The settings are same as above. The result is:

- Number of independent trials: 53,
- Number of success trials: 0,
- Successful rate: 0%
- Average value of the best individuals: 22.0

Parsimony pressure method with a dynamic parametric coefficient: The parametric coefficient sequence is {0.0f, 0.00001f, 0.00001f, 0.0001f, 0.0001f, 0.001f}, where each parametric coefficient adapts to the individuals in size of the range, [40*(i-1), 40*i] (here, i is the index of a coefficient starting from 1); the other settings are same as above. The result is:

- Number of independent trials: 24,
- Number of success trials: 3,
- Successful rate: 12%
- Average length of the success individuals: 26.5

Parsimony pressure method with a dynamic parametric coefficient is more efficient than that with a constant parametric coefficient.

2.4. Fighting with individual bloating

The tendency that the sizes of trees grow rapidly during a GP evolution is well known as the bloating phenomenon. Currently, there are several theory explanations for the cause(s) of bloating. Each of these theories depends, to some extent, on the presence of void code and the destructive nature of the crossover operation [Blbr01] [Soul98]. Although the code growth is a protective response to the destruction of crossover, the bloating phenomenon does nothing to increase the fitness of the individuals. On the other side, bloating caused the GP to quickly exhaust all of the available resources. For example, when cloning the program of *Triangle.c* with 31 test cases, if no bloat control method is utilized, on a standard PC, the evolution may terminate within 100 generations with an out of memory exception. Another hazard of bloating is that, even if a final solution is found, the solution with a massive size is generally not really useful. For example, in our trials to clone the *Triangle.c* program with 18 test cases without bloating control, the final solution found may contain 200 nodes, including much meaningless logic.

The most basic approach to bloat control is maximal depth restriction [Koza92]. Here, when a child is created by removing a sub-tree from a parent and replacing it with another sub-tree (as is done in sub-tree crossover or sub-tree mutation), if the child exceeds a maximal depth limit, the child is rejected and this process repeats until a child with an acceptable depth is created or the maximum repeating number is reached. In the later condition, a copy of the original parent takes its place in the new generation.

Combining with maximal depth restrictions, several techniques can be applied. Optional bloat control techniques include: code editing, population truncation, parsimony pressure, pareto parsimony pressure, explicitly defined introns and Sean Luke's nonparametric parsimony pressure [Lupa02]. We experimented with parsimony pressure, pareto parsimony pressure and Sean Luke's nonparametric parsimony pressure, combining with maximal depth restriction.

Amongst the diverse techniques, parsimony pressure is a very popular bloat-control technique and has some level of acceptance within the GP community compared to the other techniques. The basic parsimony pressure or constant parsimony pressure penalizes larger programs by adding a size dependent term to their fitness. The standard fitness of an individual is calculated by adding the size of the individual, weighted with a parsimony factor α, to the original fitness: F1= F0 + α *Size. Constant parsimony pressure is efficient and easy to calculate. However, two main aspects hinder the utilization of it. Firstly, it is difficult to assign a suitable value for α. In its fitness calculation, α represents how many units of size are worth one unit of raw fitness [Lupa02]. But, there is no direct relationship between the fitness value and the individual size. For different problems, α can be quite different. Secondly, the GP evolution process is not uniform, thus an α works well at the beginning phase may impede the GP converging to the final solution in the later phase. Consequently, the adaptive parsimony pressure technique, in which α is varied constantly during the evolution, is proposed.

Pareto parsimony pressure is used in multiple-objective evolutionary systems. When it is applied to reduce bloat in GP, it treats the size of individuals as the second objective that has the same importance as the first objective. Then, it aims to find a set of solutions that cannot be improved in one objective without degradation in another [Blbr01]. Strength Pareto Evolutionary Algorithm (SPEA) is an improved version of the standard pareto parsimony pressure; SPEA2 further improves SPEA in the fitness assignment.

Sean Luke later proposed a nonparametric parsimony pressure concept [Lupa02]. In his main method called double tournament, tournament selection is utilized twice. The double tournament algorithm selects an individual using tournament selection, however the tournament contestants are not chosen at random with replacement from the population. Instead, they were the winners of another tournament selection. The two tournament selections are based on fitness and size of individual respectively.

ECJ package, upon which our trials are based, implemented SPEA2 and the double tournament method. The target program is *Triangle.c* with parameter settings as before was used. However, the

results for SPEA2 are poor. SPEA2 is a complex algorithm, and it requires so many resources that within ten generations, it terminates with an out of memory exception on a standard PC. For the double tournament method, we set the parsimony tournament size as 1.3 according to [Lupa02]. However, its results are unstable and the value of fitness and length of individuals fluctuate violently. Worst of all, as illustrated by Figure 2.4.1, there is no converging tendency in its iteration. Contrast this with Figure 2.4.2, which shows the performance of the dynamic parsimony pressure algorithm.

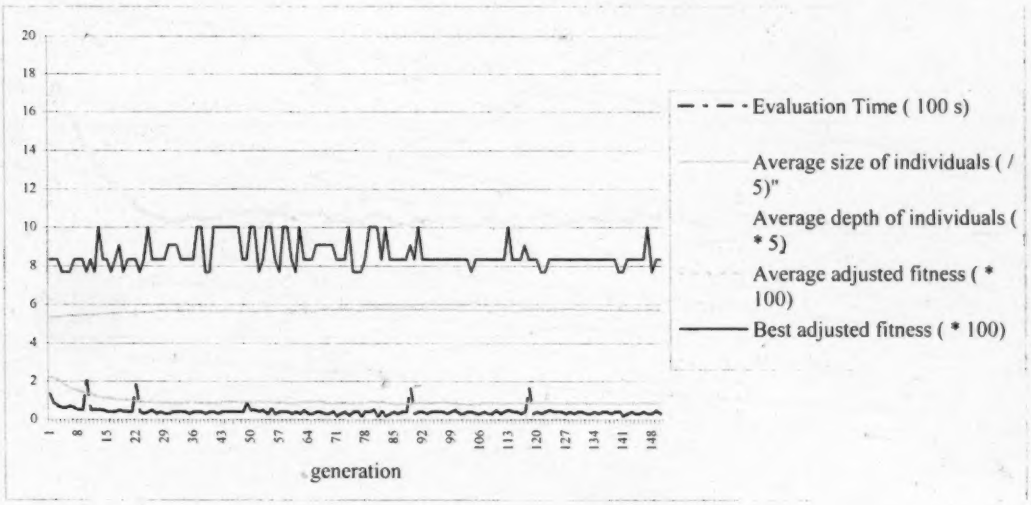


Figure 2.4.1 Performance with double tournament

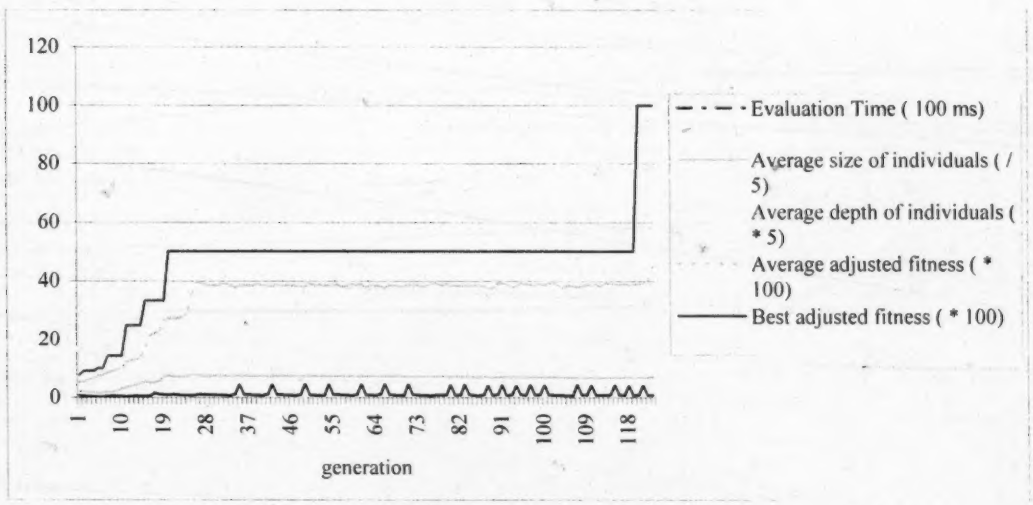


Figure 2.4.2 Performance with dynamic parsimony pressure

The lack of success for the double tournament method could be due to the value of the parsimony tournament size, which although recommended by Sean Luke [Lupa02], might not be suitable in the APGP domain.

Based on the ECJ package, we have also experimented with the constant parsimony pressure and dynamic parsimony pressure for our problem domain. The target program is *Triangle.c*. Although the constant parsimony pressure works poorly, we can achieve excellent results by applying dynamic parsimony pressure.

The dynamic parsimony pressure is an extension of the constant parsimony pressure. The key in the dynamic parsimony pressure is the variation of α. For individuals with different lengths, the parametric coefficient, α, is different. Thus the parsimony pressure is no longer linear. According to the result, the nonlinear parsimony pressure seems more efficient in adapting to the evolution process. However, GP's behavior is sensitive to the setting of the α value within array. For different problems, their settings of α value array should be different. Fortunately, our research problems focus on the APGP problem, hence, we expect that there is an α value sequence which provides reasonable results for this single domain. Furthermore, the dynamic parsimony pressure proves that a certain adaptive parsimony pressure, which guarantees effective bloat control, exists.

2.5. Representation of diverse program structures

Three programs, *Triangle.c, exact,cpp, Journey.cpp*, were selected to be cloned in our experiments. Each of them has specific characteristics. The trial emphasis for each program is different according to the program's characteristic.

2.5.1 Trials on the Triangle.c program

Same as the trials in Section 2.2, the trials in this section focus on triangle function. Detailed function description and code for this program have been given at the beginning of Section 2.2. This program is considered relatively simple. The function under estimation has n (n>=1, here n=3) inputs in the form of parameters and one output as the return value of this function. No reference parameter is used and

all of the data types are the basic data type, integer. This characteristic enables us to borrow the existing GP individual representation (which is used in the traditional GP problems such as the symbolic regression) in the program-cloning process. However the second characteristic, abundant if-then-else logic, adds complexities to our GP application. In this program, 10 out of 27 commands are if (else if) commands; in addition, nested conditional structures exist. Finally, one intermediate variable is used in the function to represent the combined effect of the inputs to the return value.

The individual representation is a tree-based structure, one tree (chromosome) is used for each individual and the terminals and return for each tree are integer. To describe such a program in GP, we defined 11 elements in the node set, 4 in the terminal set and 7 in the function set. The terminal set includes i, j and k, each represents one input parameter with the type of integer, and one integer constant with a random value. The function set includes three numerical computing operations, +, - and *, three comparison operations, =, > and < and the if-then-else logic operation, if. The if function has three children: the first one, which represents the predicate of if, is in Boolean type; the second one is in integer type and will be evaluated when the first child returns true; and the last children is in integer type and will be evaluated when the fist child returns false. In this trial, no special function is designed to represent the operations on the intermediate variable because the combination of several other operations in the node set covers the same function. Other parameters and settings are as described in the table in Section 2.2.

The trials on this program are described in the dynamic parametric method in Section 2.2. Out of the 25 trials, 3 of them achieved perfect solutions and all of the trials passed more than 25 out of 31 test cases. One of the perfect solutions is as the following:

(if (> (+ (- (* i 3) (+ k j)) k) k) (if (< i (+ k j)) (if (= k i) (if (= k j) 3 2) (if (= i j) 2 (if (= k j) 2 1))) 4) 4)

To see the internal logic more clearly, we express the above line into a C program as in Figure 2.5.1.1:

```
if ( 3 * i – ( k + j ) + k <= k)
  return 4;
else if ( i >= j + k )
  return 4;
```

```
else if ( k == i )
  if( k == j)
return 3;
else
return 2;
else if( i == j )
  return 2;
else if(k==j)
  return 2;
else
return 1;
```

Figure 2.5.1.1 Perfect solution expressed in C

Now, it is clear that except the first predication, all of the subsequent logic is reasonable. The occurrence of the first predication can be due to the misinterpretation of a particular test case(s). By modifying the test cases to be more representative, prolonging the GP evolution process or polishing the GP algorithm to be more efficient, better solutions can be found. However, for this problem, the cloned program is sufficient to deduce a totally correct version of the original program by adopting some other techniques. For example, using metamorphic relation analysis [Chtz02], we can work out that any permutation of a triple (i, j, k), causes the triangle function to produce the same result; applying this rule to the above program, the first condition can easily be removed and the missing predications, j >= i + k and k >= j + i, can be fetched.

2.5.2 Trials on the Extract.cpp program

Extract.cpp is a program for decrypting direction and hour values from a five digital number using certain rules. The code to be cloned in this trial is a fragment within the whole implementation. The function of this fragment is to check the validity of the five digits in the number, compute the value that represents the direction, which range from 1 to 8, from the first two digits and calculate the value of the hour from the digits arguments. Furthermore, the value of the hour is checked to make sure that

it is not only reasonable for represent hours in a day but also not within the special time range from 5:00 o'clock to 17 o'clock. The program fragment to be cloned is shown in Figure 2.5.2.1.

```
if (iDigit1 * iDigit2 * iDigit3 * iDigit4 * iDigit5 == 0){
    cout << "Invalid code: all digits must be non-zero." << endl;
}else{
    iExtractionHour = (iDigit3 * iDigit5) - iDigit4;
    if (iExtractionHour<0 || (iExtractionHour>5 && iExtractionHour<17) || iExtractionHour>23){
        cout << "Invalid code: incorrect extraction time provided." << endl;
    }else{
        if (((iDigit1+iDigit2) % 2) != 0 ){
            cout << "Invalid code: incorrect extraction location provided." << endl;
        }else{
            if (iDigit1 == iDigit2){
                iExtractionLocation = 1;
            }else{
                if (iDigit1<iDigit2)
                    iExtractionLocation = iDigit2;
                else
                    iExtractionLocation = iDigit1;
            }
}}}
```

Figure 2.5.2.1 Code of *Extract.cpp*

The inputs for this fragment are five integers, each of which represents one of the five digits. The outputs of the fragment include: 1) "Invalid code" messages, which represent invalid digits, or nothing indicating a success transaction, 2) the direction represented by an integer number and 3) the hour represented by an integer number. To simplify the problem, in our cloned program, we use Boolean values to represent whether the transaction is successful. This code fragment has n (n>=1, here n=5)

inputs and n (n>=1, here n=3) outputs. This code fragment equals to a function that has 7 parameters of 5 value parameters and 2 reference parameters and returns a Boolean value.

In order to symbolize three outputs for the solution, we used three trees (chromosomes) for one individual: the first tree for the hour, the second tree for the direction and the third tree for the main body of the code. The first two trees return integer values and the tree for main body returns a Boolean value. The terminals for those three trees include five variables representing the five digits, one integer constant with a random value with range from –100 to 100 and one Boolean constant with a random value. Additionally, the third tree has two more terminals: the fist two trees, which act as two automatic defined functions (ADF) [Koza92] for the third tree. However, in the fitness computation for the individual that they represent, the output from each tree is matched with the correspond output in the test cases and contributes to the final fitness.

Representing the code fragment in this way, we have divided the whole code fragment into three sub-functions each of which has 5 inputs and 1 return. Simultaneously, we have provided the cloning process with the information that the hour and direction are independent variables. Thus we reduced the searching space of GP with the cost of losing a little automation. Our experiments suggest that the information, which can be deduced from functional specification, is very helpful in searching a perfect cloning program. In addition, a sequentially performing procedure is needed. This procedure produces the cloning program for one of the independent sub-functions, delivers the result to the sub-functions that require this information, and then continues the next cloning process.

The parameters and settings for this GP application is shown in Table 2.5.2.1.

Objective:	Search a clone program that implement the same function as the code fragment in *Extract.cpp*.
Data Type Specifying:	Integer, Boolean.
Terminal Set:	Variable d1, d2, d3, d4, d5, an integer constant of random value (range: -100 ~ 100) and a Boolean constant of random value.
Function Set:	+, -, *, %, =, >, >=, <, <=, If-Bool (if-then-else logic, return Boolean),

	If-Int (if-then-else logic, return integer)
Node constraints:	Return type: tree1: integer, tree2: integer and tree3: Boolean; Variable d1~d5 are integer type +, -, *, %: with 2 integer children, return integer; =, >, >=, <, <=: with 2 integer children, return Boolean; If-Bool: all three children and return are Boolean type; not be used in ADFs If-Int: the first child in Boolean type and the other two children and return are integer type.
Fitness Calculation:	50 test cases are designed according to the code segment; an evaluation process for a individual gets input values from the 50 test cases and the output of each tree is compared with the corresponding output in test cases; score one unit per successful output; perfect score = 150, standard fitness = raw fitness = the number of unmatching cases; adjust fitness = 1/(1+standard fitness)
Hits:	The number of matching cases.
Parameters and setting:	Population number = 4000, Maximum generation = 501, Cross-Over Probability = 0.7, Mutation Probability = 0.25, Reproduction Probability = 0.05, Tournament Selection
Success Predicate:	Adjust fitness = 1.0 or Hits = 150

Table 2.5.2.1. Parameters and settings for trials of *Extract.cpp*

In our trials, the ADFs that represent the hour and direction can be easily found. However perfect solutions for the third tree, which represents the main code fragment, are difficult to be found using the parameters and settings in Table 2.5.2.1. The best individual listed here still misses 2 test cases:

Tree 1:

(- (* d5 d3) d4)

Tree 2:

(if-int (= d2 d1) 1 (if-int (<= d2 d1) d1 d2))

Tree 3:

(if-bool (>= (* 32 d2) d4) (if-bool (if-bool(< d1 (if-int (>= d2 ADF0[1]) d1 d3)) (>=(- ADF0[1] d2) d4) (if-bool (<= (% d2 (* d3 d5)) (if-int (>= d2 ADF0[1]) d1 d3)) 0(if-bool (> d5 d4) (= d3 d4) 0))) (>= (* 33 ADF0[1]) (* (* d3 d5) ADF0[1])) 0) 0)

One main reason for the poor representation for the main code fragment is in that we clone three sub-functions concurrently instead of sequentially. Consequently, the destructive effect of the crossover operation is more prominent than that in a condition with simpler individuals. In addition, the fitness calculation of the individual has to integrate the fitness of several trees (chromosomes), and a simple summation of the fitness for each tree (as in this trial) is generally unreasonable.

2.5.3 Trials on the Journey.cpp program

The background of *Journey.cpp* problem is the Collatz Problem or the 3n+1 Problem, which is described in Section 2.2. Our trials focus on the code fragment that calculates the series and the maximum value for one a_0. The code fragment to be cloned is shown in Figure 2.5.3.1.

```
int n = current;
int peak=n;
while (n != 1){
    if(n % 2)
        n = 3*n + 1;
    else
        n = n / 2;

    if (n > peak)
        peak = n;
}
```

Figure 2.5.3.1 Code fragment of *Journey.cpp*

The inputs for this fragment are two integer variables, *n* and *peak*. The outputs of the fragment are two

integer variables, *n* and *peak*. If this code fragment is in the form of a function, the function has 2 reference parameters and no return. The second characteristic of this code fragment is the loop structure.

In this trial, instead of the ADF, we used a new function (assignment), which implements the assignment function for variables, to assist cloning the calculations on variables. The individual contains only one tree and a hash-table data structure, which assist assignment operation by storing variables indexed by variable names. Thus, the reference parameters and intermediate variables are easily to be initialized, modified, reset and retrieved. The assignment operation modifies the value of a variable in that hash-table. Finally, another function, *Sequence*, is introduced to clone an arbitrary program.

This program has another characteristic -- the commonly used program structure of a loop. Hence, the loop function is introduced into the node set of GP. The loop has two children: the first one is Boolean type, which represents a loop predicate; the second one represents the loop body. The loop, loop body and the sequence function mentioned above should return no type, or return a Boolean type that represents whether the program is successfully executed.

In our experiments, none of the trials clone the whole code fragment within one GP meeting all of the 20 test cases. Then, we divided the code cloning into two steps: cloning the loop body first and then the loop structure. Among the 53 trials on cloning the loop body, 2 achieved perfect scores and 39 solve more than 36 out of 40 test cases. This two-stage process enables the system finding a perfect solution for the loop structure nearly every trial. A cloning program for the loop body with a perfect score is as follows (":=" represents the assignment function, "->" represents the sequence function):

(-> (if (> n (+ (% n n) ((% (% n 2) peak) peak))) (:= n (/ n 2)) (:= n (+ 1 (* 3 n)))) (if (> peak n) (:= peak peak) (:= peak n)))*

Translating into C, the above program can be re-expressed as shown in Figure 2.5.3.2.

```
if(n > n % 2 % peak * peak)
    n = n /2;
else
```

```
        n = 3 * n + 1;
    if (peak > n)
        peak = peak;
    else
        peak = n;
```

Figure 2.5.3.2 Cloning program translated in C language

Although the predicate of the first "if" looks strange, the cloning program is 100% correct in logic.

2.6 Conclusions and the following work

Our APGP experiment based on general-purpose GP package, ECJ, is very fundamental. Only limited techniques are tried and the target programs in our trials are simple. Nevertheless, the results of our trials seem to suggest that it is very possible to clone more complex programs or automatically implement high complexity programs that can compete with hand-written programs. Besides diverse techniques that focus on the GP evolution process, adopting a more flexible way to represent programs is very important. To carry out further research on program cloning, a particular GP package of program cloning is required.

Chapter 3

Implementation of the program-cloning package

Based on the program-cloning experiments using the general-purpose GP package, ECJ [Luke02], we designed and implemented a trial version for a program-cloning specific GP package. This package will be referred to as the program-cloning package or PCP later in this thesis. PCP provides the convenience of configuring and performing a program-cloning process for different programs/problems. Compared with the ECJ package, PCP includes more approaches to fit the GP evolution process into the program-cloning problem, thus making it more flexible and efficient than ECJ.

3.1 Architecture

The trial version of the program-cloning package was implemented in Java, version *j2sdk1.4.0_01*. The whole package can be divided into four parts: the main body of the GP algorithm, the function/terminal set for constructing a solution, specification of programs being cloned, and assistant utilities. The architecture of PCP is shown in Figure 3.1.1.

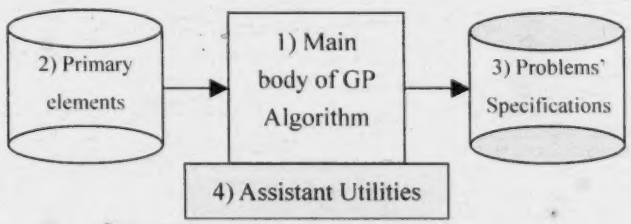


Figure 3.1.1 Architecture of PCP

1) Classes under the main directory construct the main body of the GP algorithm and implement basic structures such as nodes, individuals and populations. These classes, building the main part

of PCP, reflect most of the characters in PCP's implementation and are invariable for different program-cloning applications.

2) Classes under the *function* subdirectory provide optional function/terminal node-types that are used to construct GP's individual programs. This directory contains a whole primary-element set for the program-cloning problem; however, when carrying out program cloning for different programs, we need to carefully choose different subset of elements with respect to the characters of target problems. For example, it is necessary to include a string-copy function to clone a string-processing program, but not for a numeral-processing program. In this element set, we can conveniently add new function/terminal node-types or extend old ones.

3) A class under the *program* subdirectory describes the target programs to be cloned and the corresponding GP settings. In this thesis, such a class is called a problem specification class. In the same way that program cloning is a GP's application, cloning a particular program is a program-cloning's application. To establish the program-cloning's application, the only work is to create a specification class for the problem of interest.

   The most difficult point in defining a program specification is how to express the GP searching environment in a way that can sufficiently reflect the target problem. Generally, problems' behaviors can be reflected by data sets or by a feedback source [Kush02]. In our experiments, data sets are adopted to specify a program's GP searching context. In this way, each program-specification class defines a set of typical test cases of the target problem, each of which corresponds to an input/output case. All of the cases are selected deliberately in order to completely reflect the function contour of the target problem. The GP evaluation process is then based on how the potential solution accords with the test cases.

4) Classes under *util* are assistant utilities such as a random number generator and quick sorting tools for the population.

3.2 Implementation Features

As described in Section 1.2, program cloning is a new subject in GP's application domain. When

fitting the GP algorithm into the program-cloning problem, approaches focusing on the following aspects are devised, tested and adopted in PCP:

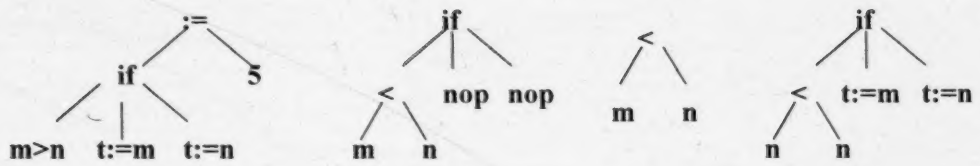
1) Strong typing
2) Exception handling
3) Input/Output handling
4) Constant
5) Dynamic fitness calculation
6) Issues concerning implementing GP in Java
7) Sub-function
8) Assigning suitable rates for crossover, mutation and reproduction
9) The number of test cases
10) Dynamic mutation probability and other implementation aspects

3.2.1 Strong typing

The individuals in PCP are represented in a tree-based structure. In order to deal with the unnecessarily huge solution space for GP searching, both Montana's Strongly Typed Genetic Programming (STGP) [Mont95] and Koza's constrained syntactic structure [Koza92] are adopted in PCP. Moreover, with respect to the program-cloning problem, we introduced new contents into the above two methods.

As explored by Koza [Koza92], for a complex GP application, the potential solution space is generally invalidly huge, making it important to reduce the searching domain by introducing restrictions that can be deduced from problems' specifications. For the program-cloning problem, the solution space can be especially invalidly huge due to the great number of optional elements in any programming language. For example, under conditions with 10 optional elements and with the maximum depth of an individual tree being 8, if no restriction or rule is added onto the solution space, even the binary full tree individuals include 10^{255} cases, most of which, however, are insignificant, but lower the GP's searching efficiency.

See the trees in Figure 3.2.1.1.



1) invalid individual 2) insignificant 3) insignificant 4) partly insignificant

Figure 3.2.1.1. Four invalid individuals (:= is assignment operation, *nop* means no operation)

In the first tree, because the constant value 5 is unreasonably assigned to an *if* structure, this assignment operation is invalid and inexecutable; the second tree, although valid in syntax, is obviously logically invalid and generally contributes very little to the GP searching process; in the same way as the second tree, the third is incomplete in logic because a reasonable root function of a general program which program cloning concern should not be a comparing operation; and finally, the fourth tree is partially insignificant in that the predication of *if*, $m<m$, is too weak to act as a branch point and leads the whole tree to retrogress to $t:=n$.

For a hand-written program, compilers depend on syntax rules to exclude the syntax invalidation; programmers use common sense to avoid the obvious logic deficient conditions; and programmers also depend on programming experiments to avoid more complex logic errors.

Excluding invalidation cases in GP's population and reducing GP's searching space, are essential to speed up GP searching process among the huge number of optional individuals. Koza's constrained syntactic structure [Koza92] and Montana's strongly typed genetic programming (STGP) [Mont95] both provide solutions for the above problem. As described in Section 2.2, our previous trials of APGP adopted both Koza's and Montana's methods, and the results proved that these two methods, complementing each other, well accelerated GP searching process in the program-cloning problem. Based on the previous experiments, the above two methods again are adopted and extended in PCP.

Within the framework of the program-cloning specific package, we can conveniently define, redefine or extend syntax structures which include more restrict constraints. In our previous experiments, the data-type checking and constraints defined for function nodes aim to keeping out

trees which are syntactically invalid. However, because insignificant or partly insignificant trees (tree 2, tree3 and tree 4 in Figure 3.2.1.1), in the same way as syntactically invalid tree (tree 1 in Figure 3.2.1.1), cannot be the target solution, they should also be omitted by the valid solution space. Consequently, PCP defines a series of constraints to keep out the meaningless or insignificant structures during individual constructing and genetic operating, and at the same time, adopts data type checking and syntax constraints as well.

In a GP application, to exclude the meaningless structures from individuals generally requires special knowledge from the application domain. Because a general-purpose GP package, like the ECJ, is independent of any application, it cannot provide a convenient way to define constraints excluding meaningless structures (as described in Chapter 2). For the program-cloning problem, the special knowledge includes general syntaxes of programming languages and common sense of programmers. PCP adopted the above knowledge to define constraints within primary elements under its *function* subdirectory (see Section 3.1 Architecture); a program-cloning's application, if necessary, can redefine or expand constraints through extending primary-element classes.

Three main constraint types in PCP are described as the following:

1) Position constraints:
   - Root set: As shown by the third tree in Figure 3.2.1.1, not every function can act as the function for root node of an individual. So we defined a root-function set and make it containing structure-controlling functions such as *seq* (sequential executing), *If-then-else* and *loop*, and the assignment function.
   - Layer checking: As described in Section 2.2 Strongly typed GP, some functions like *if* and *seq* can take only functions as children node; so, comprised in a tree structure, they must be ensured to locate at least above the lowest layer of function-nodes.

2) Possible function/terminal set of children: we adopted Koza's constrained syntactic structure and defined possible child types for each node. For example, the function *Or* can only have Boolean type children; so we define its child type set to include *Equal, Gthan* (greater than), *Lthen* (less than), *GthenE* (greater than or equal to), *LthanE* (less than or equal to), *and, Or,* and *Not.*

3) Content checking aims to reduce the insignificant structures within an individual such as tree 2 and tree 4 in Figure 3.2.1.1. For example, for a comparison function, like <, we define a constraint to make sure that different children are assigned to it.

Numerous rules and programming common senses can be adopted in the program-cloning process to guarantee the validation of individuals and reduce GP's searching space. Only a limited number of rules are abstracted into constraints in our trials, however, the results shown in the next chapter suggest that even very limited usages of the rules and programming common sense enable GP's searching efficiency obviously improved. The more complex the target program being cloned, the larger the function set is needed to construct an individual program and the more important it is to adopt problem-specific knowledge to reduce GP searching space.

From Montana's STGP, we borrowed four methods: data typing, VOID data type, local variables and errors handling [Mont95].

1) Data typing: The data typing defines data types for each variable, constant, function and function's argument, and then keeps out certain invalid individuals/structures by ensuring that data types of parents and children are consistent. Since Montana's data typing is coherent with computer languages' data typing, when PCP simulates a hand written program in a general computer language like C and java, the data typing method is definitely useful in order to exclude individuals with data typing errors from solution space.

PCP has implemented basic data types: *notype, int, double, Boolean* and *string*. The *notype* corresponds to the VOID data type in Montana's STGP and is assigned to program-flow-control functions, such as *seq*, *if*, *loop*, *assign* and *nop*. Recall that in PCP, the root of an individual tree should belong to a node-type of *seq*, *if*, *loop* or *assign*, and thus individuals in PCP return nothing. Actually, an individual program' execution results are reflected by the output variables which function in a GP's individual program in the same way as C language's reference parameters which are defined for output purpose. This usage, mentioned as acting via "side effects" by Montana [Mont95], and is one of the main differences between program cloning and conventional GP applications. Using return or output variables, PCP's individual programs are more similar to general programs. Detailed

description of the usage of output variables will be provided in Section 3.2.3 Input/Output handling.

In PCP's data typing method, terminal types such as the *input, output* and *constant* can represent variables/values of any type, and their definitions should be included in the target problem's specification class. The comparison operations like >, = and < and logical operations like *not, and* and *or* return Boolean type. The computation operations like +, -, * and / return data types same as their inputs. For example, when having two integer inputs, + returns values of integer type; when having two string inputs, + returns values of string type. All comparison operations and computation operations should accept inputs having different but compatible data types.

Data typing is applied in individual construction and genetic operation. The optional children are checked to make sure having compatible data types with their parent, and simultaneously, children's types are compatible to each other. For example, for the + function, if the first child is a float, the second one cannot be assigned to representing a string variable. Also, a crossover operation is denied if it is carried out between a sub-tree returning integer and another one returning string.

2) VOID data type: As described in the above method, in PCP, VOID data type is widely adopted by program flow controlling operations. In fact, all individual programs in PCP return VOID types and this is an important difference between program cloning and conventional GP applications.

3) Local variables: Local variable is a basic usage in hand-written programs using any high-level language. In PCP, applications can use the *input* node-type to define local variables in the same way as defining input variables. The *input* node-type contains one data-type definition and one initial value definition, and default initial values are defined for each data type. PCP's local variable method borrowed ideas from Montana's "Local Variables" [Mont95].

4) Errors handling: The error handling is one of the main characters of PCP. Different from Montana's Run-Time errors handling method, PCP's error handling can not only resolve individuals' abnormal executions like Division-By-Zero or Too-Much-Time [Mont95] but also simulate the invalid return conditions in normal programs. For example, programs, at the beginning of their codes, generally check the validation of their parameters, and if any input is invalid, a program may terminate its execution after returning an error code or message. PCP defined a special terminal node

InvalidException to simulate programs' invalidation returning conditions. Whenever the evaluation process encounters *InvalidException*, the evaluation process terminates itself after setting a particular state false; and if the expected behavior here is invalid returning, the program under evaluation scores. Detailed description of *InvalidException* will be given in the next section.

3.2.2 Exception handling

The syntax constraints and data typing exclude certain types of invalid individuals during population initializing and genetic operating phase and thus, avoid the consequent errors occurring during the evaluation process. However, many other unwanted conditions, which generally break the closure of the function set and terminal set, might appear during GP evolution. To prevent these unwanted conditions, we can carefully design particular treatment for each condition, or we can adopt a more general and easier approach, exception handling, for all of these conditions. Exception handling acts during individual evaluation process, and since the evaluation process is the most time costing part in the GP algorithm, we should use exception handling only for the unwanted conditions which cannot be excluded during population initializing and genetic operating phase. In this section, we introduce how PCP adopts exception handling to deal with undefined operations (e.g., divide by zero) and how PCP uses exception handling to simulate programs' intermediately termination behaviour.

Implemented in Java language, PCP adopted Java's inherent exception mechanism. The exception handling process is describe as the following:

1) During evaluation, in case an unexpected abnormal operation (e.g., divide by zero) occurs, evaluation on the current tree stops at the evaluation method of the last function.
2) Java Virtual Machine (JVM) throws a particular exception according to the execution situation, and the exception is caught by evaluation method.
3) The evaluation method immediately resorts to exception handling part and abandons the current evaluation results. The exception handling part analyzes the exception and assigns different scores to individuals causing different exceptions.

PCP's exception handling considers the following two conditions:

1) Program-invalid-return: In a program, it is normal to abort the execution under certain conditions, for example, in the digitals processing program *Extract.c* in Section 2.5.2, if one of the inputs values is equal or less than zero, the execution terminates after displaying an error information. PCP simulates these invalid returning conditions through exception handling. Firstly, we defined a procedure node-type, *Exp*. Being one of the primary elements, *Exp* is arbitrarily selected to assemble a possible program. During evaluation on an *Exp*, the exception, *invalidException,* which represents an invalid-returning condition, is created and thrown. The outside program which calls evaluation procedure catches the *invalidException*, and then, the exception handling part calculates the raw fitness for the current individual according to the corresponding fitness case, which should reflect the target problem. If the expected behaviour of the problem is invalid returning, the current individual scores one; otherwise it scores zero.

   In GP's application, errors such as Montana's Inversion-Of-Singular-Matrix and Bad-List-Element [Mont95] belong to the programs' invalid returning conditions and can be resolved by PCP's exception handling method.

2) Closure compensation: Closure is one of the main requirements for the function set, which states that any non-terminal should be able to handle, as an argument, any value from a candidate function or terminal [Koza92]. In order to achieve closure, particular operations concerning abnormal situations are generally defined for specific problems. For example, for the arithmetic operation division, in order to permit zero acting as the second operator, the invalid situation divide-by-zero is defined to return 1. Obviously, this method potentially can misguide GP's searching. Adding constraints to function node can also avoid invalid operation and achieve closure.

   In PCP, we tried another approach which ignored the possible invalid operations in functions' definition problem, remained them until evaluation process and then, caught and analyzed the error exceptions and scored the individuals containing them. So, in PCP, for operations such as divide-by-zero, square-root-negative and array-index-out-of-bound, instead of adding constraints or assigning a random value, we simply permit them contributing to

individuals' structures. During evaluation, exceptions, for example float-overflow, occur, and depending on Java's exception mechanism, PCP's exception handling part catches the exceptions, and according to the exception information, assigns poor scores to the corresponding individuals.

To handle the closure conflicting, in Montana's approach, a function returns null to signal that an invalid operation or a close conflict occurs; in Koza's, a function simply returns a particular value for invalid operations. By using exception handling, PCP's extended Montana's method in order to avoid misleading effects caused from the particular values. Moreover, PCP used exception handling to simulate invalid returning in a program, and the successful application on cloning *Extract.cpp* is shown in Section 4.2.

3.2.3 Input/Output handling

Individual representation is very important in GP algorithm. In program-cloning problem, the solution being represented is a normal high-level program, which includes more complex behaviors than conventional GP applications. However, the whole profile of the target problem, which is GP's evaluation environment, is expressed by input/output cases. To well reflect a problem's character and also simulate the usages of variables in a program, PCP used two special terminal types: *Input* and *Output*.

In conventional GP applications, an individual program can return one value as this individual's evaluation result or create certain side effects during its execution. The evaluation-result value or the side effects reflect how well the target individual adapts to the evolving environment and act as basis for fitness calculation. When applying GP to clone a normal program, we need to consider many inputs, outputs and their relationships for the problem being cloned. For example, *extract.c*, as shown in Section 2.5.2, has five inputs, two outputs, and one status indicating whether the execution get success. Hence, instead of calculating an individual's evaluation-value, PCP's evaluation process executes an individual program and records the evaluation results or side effects using variable terminals, which are expressed by *Input* and *Output*.

In PCP, two instance containers for *Input* and *Output* are defined respectively to hold instances of all input and output variables. During population initialization, the containers are assigned to concrete variables according to the problem specification class. At the same time, individual programs are defined to return nothing, thus all of the operations defined in a tree work around the input/output variables. Finally, the individual's fitness is calculated according to the execution result recorded in the *Output* (or *Input*) instances.

Since many program elements, especially diverse I/O operations such as standard output to screen and I/O stream operations are difficult to simulate in evolution algorithms, the program-cloning method may be unpractical or even inapplicable for certain problems. As well, in order to simulate a certain program by GP, it is often necessary to simplify the target problem. For example, the two typical usages, reading arguments from standard input and error message displaying, can be simplified as using input and output variables respectively. In PCP, the *Input* can represent functions' arguments, global variables in the target program, and values read from outside; the *Output* represents reference parameters which return values, messages being displayed, functions' return values, and global variables modified by the target program.

The *Input* terminal type is defined in a Java class, each input variable is represented in an *Input* object, and all of the *Input* objects are recorded in *Input*'s static object array – *inputs*, and besides sharing *inputs*, each *Input* object also includes information like initial values, latest values and its data type for its own input variable. During GP algorithm initialization, one *Input* object is created as an instance container holding references for all input variables. According to the problem specification class, *Input* objects are then created for input variables, and references to these objects are added into this instance container's static array *inputs*. During individual construction, from the static array *inputs*, PCP randomly selects *Input* instances as terminals to assemble individual trees. During individual evaluation, input values stored in the *Input* objects are accessed for calculations, changed by the assignment function and retrieved for consequent program execution. Because *Input* not only records a variable's initial value but also keeps recording the variable's latest values during the program execution, *Input* can also represent intermediate variables.

In the same way that *Input* functions in program cloning, *Output* is used to represent output variables. Each *Output* object records information like initial values, latest values, output values expected and the data type for its output variable. A static object array -- *outputs* is defined in an instance container and shared by all *Output* objects. When representing an output variable, an *Output* object generally adopts default initial values for evaluation, and the default values are determined by the output variable's data type. For example, an integer output variable has a set of initial values zero, and a Boolean output variable has a set of initial values false. During GP algorithm initialization, one *Output* object is created as an instance container holding references for all output variables. According to the problem specification class, *Output* objects are then created for output variables, and references to these objects are added into this instance container's static array *outputs*. During individual construction, from the static array *outputs*, PCP randomly selects *Output* instances as terminals to assemble individual trees. During individual evaluation, output values stored in the *Output* objects are accessed for calculations, changed by the assignment function and retrieved for consequent program execution. After an individual program is executed, every output variable is checked to see whether its latest value agrees with its output value expected and/or how close the two values are, and this comparing result is then used to calculate the target individual's fitness.

Input and *Output* work in the same way, and by carefully expanding any one of them, a common class can be created to represent all variable usage conditions in a problem.

3.2.4 Constant

Being another type of terminal node, a constant works in a quite different way from a variable terminal. As described in Section 3.3.3, a variable terminal is a predefined variable that is evaluated using fitness cases defined in the problem specification class, and contributes to a parse tree through the way it is combined with other functions and terminals. In contrast, a constant represents a concrete value randomly selected within a range determined by the problem, and contributes to a parse tree not only through the way this constant is combined with other functions and terminals but also by its value.

To simulate a normal program, three types of constants are adopted in PCP:

1) The problem specific constant values, which are defined in the problem specification or are well known constants in the problem domain. For example, π and e are well known mathematic constants.
2) Simple constants that are frequently adopted in programs and generally act as threshold values. For example, 0, 1, –1, true, false and null.
3) Other values, which relate to the problem's implementation details and are generally hard to deduced without deep understanding of the target problem.

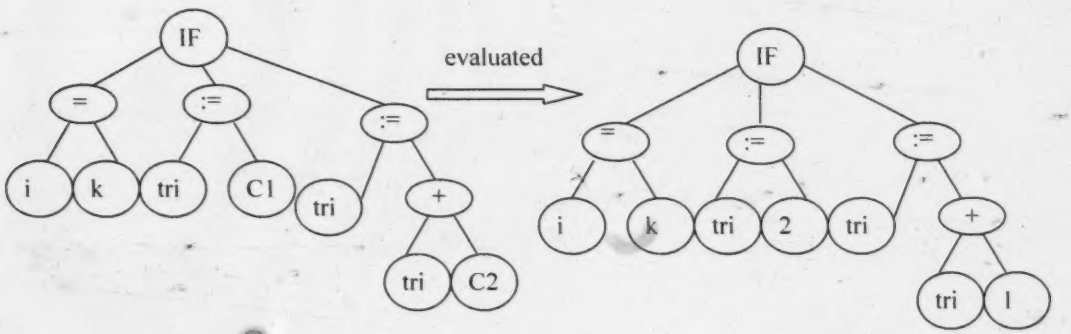
To represent the arbitrary values that may appear anywhere in GP's individual trees, Koza introduced the special terminal type calling *ephemeral random constant* [Koza92]. During population initialization, if the ephemeral random constant is used to act as an endpoint in an individual tree, a concrete value with a specified data type in a specified range is randomly created and assigned to that point. Once produced and inserted into an individual program, the value remains fixed. Furthermore, Koza defined three ways that constant atoms can enter a problem:

1) Automatically produced by GP for simple constants, for example, the little sub-tree *X/X* deduces constant 1;
2) Created most of the constants using ephemeral random constant; and
3) Explicitly included a set of particular values in GP.

Note that the second and third approaches exactly correspond to PCP's third and first types of constants respectively; the first approach, as well, is adoptable for program cloning to provide simple values that are frequently used in a normal program.

In PCP, the way to manipulate constants for an individual tree is devised with the background of program cloning; however, PCP's approach mostly agrees with Koza's method of producing constant terminals, especially the concept of ephemeral random constant. In PCP's constant manipulation method, firstly, for constants, a *Constant* class was defined as a node type. Besides basic elements applicable to all node-types, the *Constant* class includes a variable *range* for setting the constants' range concerning the target problem, and a variable *specials* for explicitly including a set of particular

values which are defined by in the application's problem specification class. To choose an arbitrary value, a certain possibility (10% in our trials) is assigned to elements in *specials*, while the left opportunity is shared by values within the range. During individual construction, *Constant*, along with other node-types, is randomly selected for end nodes to compose an individual tree. A constant terminal itself has no data type or can have any data type; in PCP, a constant terminal's data type is assigned during individual initialization and determined by its context in a parse tree. For example, in the tree 1) in Figure 3.2.4.1, C1's data type is determined by its brother, *tri*. In Section 3.2.6, detailed explanation about data types will be given.



1) A newly created parse tree

2) An evaluated parse tree

Figure 3.2.4.1. Parse tree with constant terminals (, where"C1, C2" represent two constants)

Before its first evaluation, a constant node has no value, which is shown by tree 1) in Figure 3.2.4.1. Then, during evaluation, if a no-value constant node is encountered, *evaluate* method defined in *Constant* class functions; and according to the constant node's data type, a random value is selected either from the constant range or from the *specials* array, and assigned to the node, as shown by tree 2) in Figure 3.2.4.1. After it evaluation, a constant node gets a value, and this value is fixed throughout the subsequent evolution process, and through crossover operation, this value is allocated and functions in diverse combination forms.

Compared with Koza's constant handling approach, PCP's method has two different usages: 1) Some frequently used constants like 0, 1, -1, true and false are added into the *specials* array to give them more opportunities to compose a parse tree. 2) Instead of assigning values to constant nodes during individual initiation, PCP creates and assigns random values to constant nodes during

evaluation, and thus saves the assigning operations on unreachable constant nodes.

3.2.5 Dynamic fitness calculation

Fitness is the basis of GP selection and the root force of evolution. When simulating natural selection process, it is kernel for GP to reflect how well an individual fits the problem environment through fitness measurement. Most of the GP implementations rate fitness with an explicit fitness measure for each individual in the population. In PCP, an individual's fitness is calculated through the following steps:

1) In each problem specification class, a set of fitness cases is defined as the evolution environment for population. To compose a small set of samplings for the entire domain space, the fitness cases should be selected deliberately in order to reflect the whole contour of the program being cloned. Otherwise, if we simply use randomly selected fitness cases, in order to reflect every character of the target problem, a much larger number of cases are needed. In PCP's trials, the number of fitness cases ranged from 30 to 90, the more the fitness cases the more time cost in GP. A too small set of fitness cases may causes immature convergence, and the results of PCP's trials (see Section 4.1) showed that for a simple programs-cloning problem, GP evolves well with 50 to 60 fitness cases.

2) Under each fitness case, each unevaluated individual program is executed, and the execution results for this program are recorded in *Output*'s objects (as explained in Section 3.2.3). For each individual, its execution results are then compared with its anticipated results, and according to how coincident the real results are with the correct results, this individual gains a certain score.

Generally, for GP's applications which search for calculation formulas, the above score is measured by the absolute difference between the anticipated results and the real results. However, for other problems like the artificial ant, an individual scores one or zero respectively for each case that is satisfied or not. In PCP, considering that the programs being cloned may focus on diverse problems, we adopted both of the above approaches for fitness calculation, and a user can define in the problem specification class which one is suitable for the target problem. Finally, the sum of scores reflects how well the individual fits the problem environment or can solve the problem. A pseudo code for

individual fitness calculating is shown in Figure 3.2.5.1.

```
For all individuals { | individual i|
    For all fitness case{ |case j|
        Result = evaluate (individual i)
        If problem type 1
            For all result variables{
                Score of individual i += | anticipating result of case j – Result|    -- (1)
            }
        Else if anticipating result of case j == Result      -- (2)
            For all result variables{
Score of individual i += 1
}
    }
}
```

Figure 3.2.5.1 Pseudo code for individual fitness calculating (where formula (1) and (2) focus on the above two raw fitness measurements respectively)

3) With Koza's fitness calculation method [Koza92], the score obtained in step 2) is called raw fitness, i.e., *r(i)* where *i* represents an arbitrary individual. In order to reveal the subtle difference among individuals' fitness, we need to translate raw fitness values into adjusted fitness values. The adjusted fitness a(i,t) is computed from the standardized fitness s(i,t) as: $a(i,t) = 1 / (1 + s (i , t))$. An individual's standardized fitness equals to its raw fitness in case formula *(1)* in Figure 3.2.5.1 is adopted and otherwise, equals to *full mark – r(i),* where the *full mark* is calculated during GP initialization.

Based on fitness values, individuals are ranked and selected and take part in genetic operations, and new offspring occur, and superior genes enter into the next evolution loop.

One big problem in the genetic algorithm is that the searching space is not uniform. Even with the fitness cases well representing the problem details, peaks and coves in the solution space hinder the

evolution approaching to the final program. In program cloning, since the target program is made of a series of compositive instructions that have no inherent relationship, an individual generally includes only one or several segments of the final program. A common unsuccessful case in our trials is that the population's best individual which missed only one or two fitness cases seemed to be blocked forever during GP's evolution. The un-uniform solution space causes some individuals to evolve quickly while missing some aspects of the problem; and meanwhile, individuals which meet the minor one or two fitness cases have no opportunities to enter the genetic operations due to low fitness.

To solve this problem, PCP adopted an improved measurement for fitness calculation, which assigns higher priorities to individuals meeting new fitness cases. To do so, we introduced two variables into the evaluation method: *newHits* which records how many new fitness cases are met by the current individual, and *missed* which records indexes of the fitness cases that have not been covered by the population. An individual's final fitness, which is the basis of evolution selection, is calculated by the following formula:

$$Fitness = newHits / missed.length * (bestSolution.fitness - currentSolution.fitness) * w + adjust\ fitness,$$

where *bestSolution* is the individual who possesses the highest fitness until this point, *currentSolution* is the individual currently being evaluated and w is a constant value acting as the weight of the amendment part (in PCP, w is 50%). With this fitness measurement, the more new cases an individual meets, and the fewer cases the whole population misses, the more opportunities the individual has for surviving. And the difference between the current individual's fitness and the best solution's fitness acts in the measurement to reduce the difference between their selection opportunities. The usage and consequent result of the improved fitness measurement is explained with the sample in Section 4.1.

3.2.6 Issues concerning implementing GP in Java

The most popular and mature individuals' structure for GP is parse tree, and how to manipulate parse tree structures and carry out evaluation on them is the kernel in GP's implementations. LISP is a common language for GP's implementation. One S-expression in LISP is a structurally ready parse tree however expressed in a linear fashion: each element of terminal set is a variable or constant value

in LISP; and each function is expressed using a bracket pair, the identification and correspond sub-trees. So it is straightforward to generate and evaluate LISP individual programs. However, LISP is not the only language for GP's implementation. Pursuing efficiency and flexibility, people have implemented GP in diverse languages, such as C, C++, and Java. Due to the successful experiments with ECJ GP package (in Java) for APGP (see Chapter 2), we implemented PCP in Java language.

PCP adopted the parse tree as its individual structure, and the tree-based manipulations include: 1) node representation and evaluation, 2) individual representation and evaluation, and 3) genetic operation.

1) Node representation and evaluation:

As described in Section 1.1, GP has two types of primary elements: function and terminal. In a tree-based GP package, functions work at a parse tree's internal nodes, and terminals work at a parse tree's leaves. PCP expressed nodes based on the pointer-base approach, in which each node in a tree used pointers/handles pointing to its children and parent. In PCP, besides parent and children, a node's attributes also included node-type, value and value's data type. Each node-type expressed to a primary element (a function or a terminal), and the node-type determined a node's other contents. For example, a node with node-type of + has to have one numeric value, one numeric type such as integer or float, two numeric children and a parent which accepts numeric input at this node's position. In PCP, the node-types (or function set and terminal set) experimented with in our trials are defined in Table 3.2.6.1:

+, -, *, /, %	In function set, arithmetic operations
>, >=, =, <=, <	In function set, relation operations
And, Or, Not	In function set, logical operations
Input	In terminal set, represents input/intermediate variables, see Section 3.2.3
Output	In terminal set, represents output/intermediate variables, see Section 3.2.3
Constant	In terminal set, represents constant values, see Section 3.2.4
If	In function set, represents if-then-else program structure

Loop	In function set, represents loop program structure
Seq	In function set, represents sequentially performing program structure
SubFunction	In terminal set, represents sub-functions, see Section 3.2.7
Nop	In terminal set, represents no-operation
Assignment	In function set, represents assignment operation
Exp	In terminal set, represents intermediate return, see Section 3.2.2

Table 3.2.6.1 Main node set defined in PCP

In any programming language, the language components are far more than what are listed above. However, restricting the size of the function or terminal set is one of the keys for successful GP searching. In a particular GP application, elements in the function set and the terminal set can be deliberately selected to fit the target problem or the problem domain. In the program-cloning problem, although we try to simulate an arbitrary program of any purpose, the large number of language components forced us to narrow the node-types to a small set concerning the particular program being cloned. We assumed that by analyzing original programs or similar programs in the same problem domain, using computer-language syntax rules and operating on a friendly GUI, program cloning's users can conveniently determine and deploy node-type set for a particular problem.

In PCP's implementation, we defined the class *NodeType* for node-type, and this class includes an abstract *evaluate* method and utilities performing data typing and syntax constraints checking according to the rules defined in each concrete node-type class. A concrete node-type class extended *NodeType* and generally included the following information: node-type name, can or not be a root node, children types, special constraints or rules and most of all, the evaluation process.

For example, the *Equal* included information as following:

- *name = "==";*
- *canBeRoot = **false**;*
- *childrenName = **new** String[][] {*

{"Add","Sub","Mul","Div","Mod","Input","Output"},{"Add","Sub","Mul","Div","Mod","Const","Input","Output"}};

- *public Boolean constraint (Node parent, Function son, int position, int depth){...}*
- *public Object eval(Node[] children,int caseIndex,Node node) throws InvalidException {...}*

When *Equal* is used in composing an individual, the super class *NodeType* makes sure that nodes with type of *Equal* are not at a root position, only node-types whose names are listed in children types can be used to compose children nodes, and children's data-types are compatible. During individuals' constructions and genetic operations, *Equal*'s constraint method is executed checking node-types and content of children to make sure that the operation between the two children are significant (as described in Section 3.2.1). During individuals' evaluation, *Equal*'s *evaluate* method is executed calling children's evaluate procedures to get their values and carrying out comparing between the values and returning a Boolean result.

Special node-type classes like *Input*, *Output*, *Constant*, *Exception* and *SubFunction* included more complex usages, and are discussed in detail in other sections in this chapter.

2) Individual representation and evaluation:

Based on the node representation approach, PCP's individual representation is simple. Beginning from randomly selecting a root node-type and creating the root node, the individual construction procedure expands root node by finding suitable children for it, and simultaneously sets parent and children's pointers to each other. The children nodes are further expanded and this node-expanding process continues until all nodes newly created are terminals. Hence, a parse tree can be represented by an instanced root node and, tracing the root, access every node. An individual's other main elements include: an evaluated-or-not indicator, an adjusted fitness and a raw fitness.

The evaluation on an individual begins from root node evaluation, which then post-order-scans the root's parse tree and at the same time invokes children evaluation. GP's one individual evaluation is to execute the individual program, entering from one entrance, which is initialized by fitness cases, and outing by many exits, from which fitness is calculated. To calculate the fitness, the execution results recorded in *Output* class are compared with expected results predefined in the problem specification class. The detailed fitness calculation method is described in 3.2.5.

3) Genetic operations

As described in Section 1.2, GP's genetic operations mainly include crossover, mutation and reproduction. Together with fitness-based selection, they establish the kernel part of GP algorithm. Among the three genetic operations, crossover is generally thought to be the most contributing operation, and in contrast, mutation is generally ignored [Koza92]. However, our trials show that mutation also plays an important role in an evolution process, for example, a subtle modification of the mutation rate results in totally different GP behaviour.

Depending on fitness-based selection, genetic operations force the evolution toward environment fitting direction and thus distinguish genetic algorithms from random searching process. To carry out fitness-based selection, PCP calculated adjusted-fitness for each individual and adopted the traditional tournament selection with a size of 7. As explored by trials cloning *Triangle.c*, the success rate of GP searching with a tournament-selection size of 7 is 3 times bigger than that with size of 2.

The crossover operation begins from fitness-based selecting two parent individuals; then, on each parent tree, an operation node is randomly selected; and exchange is carried out between the two sub-trees rooted at the nodes being selected. Crossover operations on nodes of different positions in a parse tree cause different levels of exchanging; Koza has shown that internal nodes should be given more opportunities than terminal nodes for genetic operations. Generally, to select an operation node, a certain percentage is used to determin the possibility that internal nodes are likely to be selected. In PCP, this internal node percentage adopted Koza's experiential value, 90%, which enable around 90% genetic operations are carried out on internal nodes and 10% on terminal nodes.

Not in all GP application, the root node can be selected to carry out crossover, which results in combining a whole tree into another one. However, in PCP, root nodes take part in crossover due to the consideration that simple trees having only 2 or 3 layers are liable to be accordable with subtle instructions in a program, adding such a simple tree to another one or combining a series of simple trees together is liable to approach the target program. For example, when cloning the *Triangle.c*, simple tree like *tri := 4*, and *tri := 0* are definitely component in the target parse tree, thus adding these instructions into other promising individuals can accelerate the evolution.

After exchanging points for parent individuals are identified, compatibility checking is carried out

on them. Data typing and syntax constraints are adopted to exclude invalid individuals. For example, in parent tree *A*, a node *a*, whose parent is node *aa*, is selected to exchange with node *b* in parent tree *B*. One offspring is created based on tree *A*, through replacing the sub-tree rooted at node *a* by the sub-tree rooted at node *b*. Firstly, the depth constraint must be satisfied: the depth of the offspring to be created must be less than the predefined maximum depth. Then, root checking is performed: in case *a* is the root node in tree *A*, the node-type of *b* must belong to the root node-type set. Then, *b*'s node-type must be a possible children node-type of the *aa*'s node-type. Also, all of other constraints defined in *aa*'s node-type must be satisfied by *b*'s type. Finally, the basic data type of *b* must be compatible with *aa*. After the compatible checking, exchange between sub-trees is implemented by simply setting pointer/handle of the parent node to point to the substitute sub-tree, and then setting the individual's evaluated mark to false.

The mutation operation introduces random changes in population, and PCP depends on mutation to increase diversity. The single parent and target node to carry out mutation operation is selected by the same methods used in crossover operation. Once mutation node is located, a sub-tree rooted at the mutation node is built up in the same way as constructing individuals. To prevent individual bloating, the depth of the new offspring is limited to the predefined maximum mutation depth. Finally, the evaluated mark for the new individual is set to false.

When carrying out crossover, mutation or individual construction operations, it is not guarantee that an individual is created successfully. During crossover operation, failing of the compatible checking results in dumping of the current crossover. During individual construction and mutation operations, if the child type which is randomly selected, cannot pass data typing and syntax constraints, the selected type is abandoned; and in case after a certain number of trying, still no suitable child type is found, the individual under construction is deemed malformed and discarded.

To implement mutation and crossover in Java, one important issue is how to clone a parse tree object. For example, in crossover operation, the offspring are two combing structures of the parents. Both the main tree and the sub-tree should be copied into the offspring, and not only nodes but also pointers should be duplicated. In PCP, both the *Individual* class and the *Node* class have their *clone*

method respectively.

3.2.7 Sub-function

To solve a complex problem, it is natural to adopt divide-and-conquer strategy: decompose the whole problem into several simple ones, solve each of the sub-problems, and combine the solutions for sub-problems to solve the whole problem. This method is a very common usage in software programming, and each sub-problem corresponds to a subroutine in a program. Using subroutines can simplify the target problem, introduce flexibility in a solution and improve implementation efficiency. Based on divide-and-conquer strategy, for program cloning, we devised the sub-function method, and this method simulated the subroutine usage in a program, reduced the target solutions' complexity and reduced the necessary primary elements for constructing each sub-function trees.

In conventional GP applications, automatically defined function (ADF) is a popular method to carry out divide-and-conquer strategy. When utilizing ADF mechanism, GP process searches a main tree concerning the whole problem and at the same time, a number of sub-function trees concerning the sub-problems. The element set for the main tree and function trees are different according to their particular problems; however, the element set of the main tree must include the special operation ADF, which performs calling to the sub-functions. During evaluation, the execution entrance is the main tree's root, and through ADF nodes the sub-functions are executed.

However, as shown in Section 2.5.2, ADF does not work well for the program-cloning problem. Even to cloning a simple program as *Extract.cpp*, the best solution of the main body was far from ideal. The problem is that the searching for the main tree and the sub-function trees are simultaneous, and the evaluation process and fitness measurement are carried out for the main tree and the sub-function trees together, and thus one fitness value is used to reflect not only how the main solution tree fits the main problem but also how the subroutines fit the sub-problems. When applying GP to the program-cloning problem, since an individual's structure is more complex than that in a conventional GP application the above problem is more prominent and serious.

Different from ADF, our sub-function method search solutions for main tree and sub-function

tree(s) consequently instead of concurrently and thus greatly simplified the searching process. The concrete method is described as the following:

1) A new node-type *SubFunction* is defined to represent the action of sub-functions calling. The *SubFunction* class contains a class-variable *subFuns*, which is an array records all callable sub-functions, and each of these sub-functions is a *SubFunction*'s object. When building up a main tree, *SubFunction* may be selected as a terminal node-type, and then a concrete sub-function in *subFuns* is randomly selected to instance the current node. During the main tree's evaluation, in case *SubFunction* is encountered, the expected output of the corresponding sub-function, which is predefined in the sub-function's problem specification class, is returned.
2) To describe a program being cloned which adopts the sub-function method, more than one problem specification class should be defined. The specification for each sub-function is defined in the same way as that of any simple program (see Section 3.1); the specification for the main body of a program contains all class names of the sub-functions.
3) To start the program-cloning process, the algorithm initialization process firstly checks the main problem's specification. If sub-functions exist, the target of the initialization/program-cloning process changes to each sub-function one by one. After program cloning for every sub-function is finished, the main function is initialized and cloned.

Through our sub-function method, we separate a big program into several independent simple programs. Compared with traditional ADF method, our method turns the black box of the solution a little transparent and at the same time, sacrifices a certain degree of automation in GP searching. By defining more than one program specifications, we mechanically cut the big problem into pieces, and introduce some problem-specific knowledge into the GP searching process. The operation of dividing the whole problem's specification into several sub-function specifications can be very simple by employing friendly GUIs. But this method generally requires the users to possess a certain degree of understanding to the target problem. However, as shown by the example's results in Section 4.2, the sub-function method improves the efficiency of the program-cloning process significantly.

3.2.8 Assigning suitable rates for crossover, mutation and reproduction

While reproduction and crossover operations are approaches to increase the converging tendency, mutation operations introduce diversity to a population. Different from that in the genetic algorithm, crossover operations in GP exert a counterbalancing pressure away from convergence; so it is generally believed that mutation operations are not primary genetic operations and can be ignored in GP [Koza92]. However, Banzhaf has proposed totally different idea [Bafn96], and it has also revealed by our program-cloning trials that mutation operations lead an important role in GP searching processes (see the example in Section 4.1), and with different mutation rate, the behaviors of GP searching are quite different.

We have tried diverse proportion settings among the crossover, mutation and reproduction operations, and Table 3.2.8.1 shows some of the settings in our trials:

Crossover percentage	*Mutation percentage*	*Reproduction percentage*
20%	*1%*	*79%*
20%	*20%*	*60%*
20%	*40%*	*40%*
90%	*1%*	*9%*
90%	*20%*	*0%*
90%	*40%*	*0%*

Table 3.2.8.1 Genetic-operation proportion

During cloning *Triangle.c* (see Section 4.1), the trials on the first three proportions turned out with poor results due to the low probability of crossover operations. With the crossover percentage 90%, the successful rate of GP searching got improvement; however, as shown in Section 4.1, under certain settings, with the mutation rate 40%, 7 out 15 trials obtained perfect solutions (solutions which meet all fitness cases). In the last two cases, reproduction is totally omitted, and furthermore, not all individuals newly created from genetic operations have opportunities entering the next generation. Instead, only individuals have higher fitness can enter the next evolving loop. For example, with the crossover percentage 90%, the mutation percentage 20% and a population size of 4000, 400 * 110%

individuals are created and 10% of them, whose fitness rank at the lowest 40 (we call them redundant offspring), are discarded. In this way, we actually introduced elite strategy, which enhanced the converging effects and thus balanced the diversity effects introduced by mutation operations.

3.2.9 Number of test cases

To measure how well an individual fits the target problem, an executing environment or evaluation environment should be established. Generally, the executing environment can be constructed using test cases of the target problem. These test cases are called *fitness cases* in GP algorithm and they are adopted in our program-cloning experiments. However, as mentioned in Section 3.2.5, using fitness cases to simulate a problem's function behavior does not always work well, and the key is to choose suitable fitness cases: not only the number of them but also their distribution within the problem solution space is crucial.

Too few fitness cases may miss some aspects of the problem, guide the evolving to wrong direction and lead the population to un-mature converging. On the other hand, too many fitness cases cause a waste of computer resource. Depending on different problems, the numbers of fitness cases should be different. In the program-cloning problem, different programs being cloned require different number of fitness cases, and a user should define the case number in the problem specification. Possible approaches to determine the suitable case number have been studied and prompted [TeAn97] [GiTo02]; however, considering the handleability, we did not adopt them in PCP.

A suitable distribution of the fitness cases is important in order to clearly reflect a target problem's contour. Borrowing the method from software structure testing, we introduced the decision-table into fitness cases determination. First, predications for each branch are deduced from the original code to be cloned or from the problem specification; a true value table is then applied to those predications, and according to the problem, unreasonable combinations of the predications are omitted; finally, for each combination, a test case is deduced, and all of the test cases construct individuals' executing environment (see the example in Section 4.2).

In the same way as software testing, fitness cases deduced from a decision-tree might still miss

some characters of the target problem; nevertheless, more advanced method in software testing for manually creating test-data can be adopted for fitness cases determination.

3.2.10 Dynamic mutation probability and other implementation aspects

Besides the implementation characters described above, other settings/methods used in PCP are listed as the following:

1) To restrain individual bloating, PCP adopts dynamic parsimony pressure algorithm, which is experimented in APGP process with ECJ (see Section 2.4).
2) PCP always permits the best individual for each generation entering the next generation.
3) In PCP, individuals' roots can be chosen as genetic operation nodes.
4) When selecting nodes for genetic operations, PCP assigns 90% opportunities to internal nodes and 10% to leaf nodes.
5) Although bringing no obviously improvement on GP's behavior, dynamic mutation probability was still adopted in PCP. The following paragraphs will briefly introduce this method.

PCP's applications suggested that mutation operation could lead an important role to GP's success. Since mutation introduces random changes to individuals' structures and consequently weakens the convergence, it is used to adjust the population's diversity and regulate GP's converging speed. During GP's un-linear evolution process, mutation contributes to population's evolving differently in different phases. Dynamic mutation rate is widely used in the genetic algorithm; self-adaptive mutation operators are well-known within evolutionary computing [Buht00]; we assume dynamic mutation rate can help also improve GP's searching efficiency.

Our assumption is based on the results of our trials. As shown in Figure 3.2.10.1, during the early period of evolution, periodically, the unduplicated individual number (simply reflecting the diversity of population) decreases after the population evolves a certain number of generations. At the late phase of the evolution, a set of mature individuals seem to blocked there permanently missing only a very few number of problem conditions. Through dynamic mutation rate, we hope to maintain a high diversity and provide mature populations more opportunities to mutate to the target solution.

Therefore, during evolution, instead of decreasing mutation rate little by little to speed the population converging, we kept adjusting mutation rate according to the population's diversity.

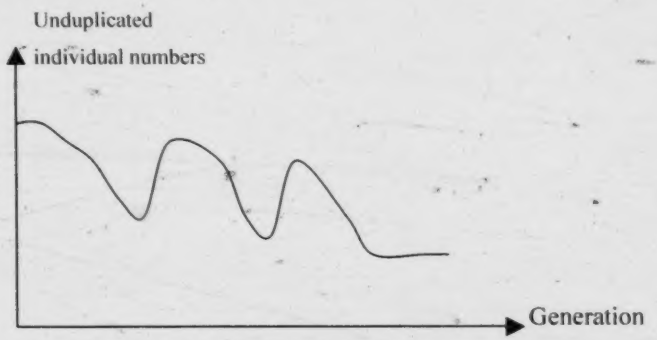


Figure 3.2.10.1. Diversity of population

In PCP, the mutation rate is recalculated after each 10 generations by the following formula:

$$Pm = R1 - (diversity - R1) * (R1 - R2) / (1 - R1),$$

where *Pm* represents mutation rate for the current generation, *R1* and *R2* are the upper limit and lower limit for mutation rate that are generally adopted, i.e., 20% and 0.5% respectively, and *diversity* is calculated by: *number of un-duplicated individuals / population size*.

Chapter 4

Experiments with program cloning using program-cloning package

In this chapter, trials on five typical problems are described to display our program-cloning experiments with PCP. These five problems have different levels of complexity, and they are typical due to their particular usages, structures, interfaces or problem functions. A simple introduction of them is as the following:

1) *Triangle.c*: *Triangle.c* is a representative program due to its complex branch structure, and many literatures use it as a sample [Jorg02][Mims01]. We have experimented with cloning it using ECJ package, and the result of cloning *Triangle.c* using PCP shows that PCP, implemented through a series of special methods as described in Chapter 3, is much more efficient for program cloning than a general GP package like ECJ.

2) *Extract.cpp*: We choose it to illustrate the usage of sub-function. Previous attempting to clone *Extract.cpp* using ECJ adopted ADF and failed to find a program for the main body of the problem. In our new experiment, this deficiency is overcome by using the Sub-Function method (see Section 3.2.7).

3) *Journey.cpp*: It is special for the loop structure. Our experiments involved in cloning *Journey.cpp* using ECJ package indicate that the loop structure is especially hard to be manipulated in GP algorithm. We attempted to clone the loop structure again with PCP, and although no much improvement is achieved, our study reveals that the difficulty of using GP to clone loop structure is due to the algorithm itself.

4) *NextDate*: This problem and the following *Commission* are cited from Jorgensen's *Software Testing* [Jorg02]. It is more complex than the above three problems. Two sources of complexity exist in this function: the complexity due to the relationship between inputs and outputs, and the

internal logic complexity.

5) *Commission*: It is the most complex problem among these five samples. As a practical commercial-computing problem, it contains a mix of computation and decision-making [Jorg02].

In the following sections, we will describe our experiments with the above programs one by one. Each section contains four parts: a program introduction, detailed approaches for applying the program-cloning method, results and analysis. For the problems which have been experimented using ECJ, we will also compare the approaches and results in different GP implementations.

4.1 Triangle.c

Triangle.c is the code for the problem of Triangle Classification. The purpose of the problem is to classify a random triangle into scalene, isosceles, equilateral and non-triangle according to three edges. As shown in Figure 2.3.1, with only 30 lines of code, the *Triangle.c* has 10 branches. However most of the decisions contain only a single condition, and the maximum depth of its branch nesting is two; and with three independent inputs (no mutual computation among them), and one output with four optional values, the problem's internal logic is relatively simple. In addition, only a small number of syntax components are used, and no complex manipulation of intermediate variables is expected.

Before carrying out program cloning, we can set up an idea solution in mind for using program cloning to solve the triangle problem, and one idea solution is as the parse tree in Figure 4.1.1.

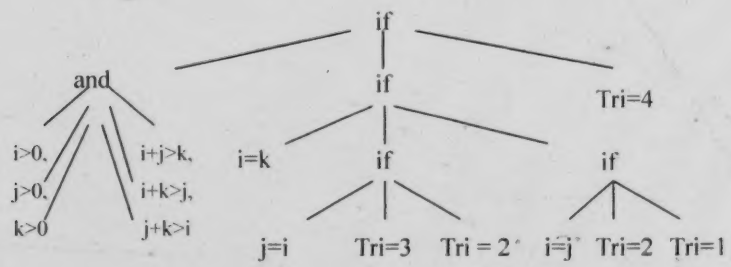


Figure 4.1.1 A solution tree for triangle, where i, j and k represent three edges and Tri is the output with the values 1,2,3, and 4 representing scalene, isosceles, equilateral and non-triangle respectively

The first major step in preparing to use genetic programming is to identify the set of terminals [Koza92]. For the program-cloning problem, we have defined three basic terminal types, *Input, Output* and *Const* in PCP. Generally, all of the basic terminal types are necessary for cloning a general

program. However, to clone different programs, these terminal types should be instanced into different contents.

The *Triangle.c* has three integer input variables: i, j and k, thus we defined them in triangle's problem specification class, and during GP initialization, they are initialized as three optional I*nput* types. In the same way, *Triangle.c*'s only output, *Tri*, is defined in the specification class and then initialized as an *Output* type during GP initialization. Corresponding to input/output pairs, a number of fitness cases are designed and specified in the specification class.

Constants are necessary in this problem. Considering that the return of *Tri* is within four values: 1, 2, 3 and 4, which represent four triangle classes and have no numerical meanings, we can include these four numbers in the particular constant set -- *specials* (see Section 3.2.4). Since these four numbers are deducible from the problem specification, we can add this special knowledge into GP searching automatically. The number 0 acts in triangle as a restriction of triangles' edges, thus it is also included in *specials*. As shown by the parse tree in Figure 4.1.1, mathematic calculation is not a main topic in this problem, thus the usage of random value is very limited. Hence, *specials* plus the typical values for programming (the second constant type described in Section 3.2.4) are enough to construct the possible solutions. However, giving the GP searching more solution space for irregular solutions, we also adopted random value generation providing integers, and without lose the generality, the range of the random integers is defined from 0 to 10.

Besides, the terminal, *nop*, which means no operation and functions in branch control structure, is included in the terminal set. In summary, the possible terminals for constructing clone program of *Triangle.c* includes: i, j, k, Tri, 1, 2, 3, 4, 0, -1, true, false, random values within the range of [0,10] and *Nop*.

The second major step in preparing to use GP algorithm is to identify the set of functions. From triangle's specification and code, we can see that *Triangle.c* is relatively simple and does not need many primary elements to construct its target program. For example, in the original program, there is only one arithmetic operation, addition. However, to make the searching process general and also avoid excluding the diversity of the solution programs, we adopted the primary function set defined in

Table 4.1.1. Including arithmetic operations, logical operations, comparison operations and program structure controllers, the function set for *Triangle.c* simulation has the following function types: +, -, *, /, %, >, >=, =, <=, <, And, Or, Not, If, Seq, and Assignment (see Table 4.1.1 for detailed explanation).

Loop is a basic program-controlling structure in any computer language, however, as long as it is not an absolutely necessary element to compose the solution program, we should always avoid including it into GP's function set. *Loop* operation is very time costing because the predications of *Loop*, which are random created, may cause abundant meaningless iterations, and then, the evaluation process has to spend many more times evaluating the loop body, and worst of all, many *Loop* can never terminal themselves without external interruptions. As a heuristic algorithm, GP needs large population and tries diverse opportunities in its searching process. But, when using loop to compose potential solutions, GP will spend almost all of its time manipulating *Loop* structures and hardly maintain its normal searching process. Even using execution-time limits for *Loop* operations, the basic problem described above cannot be solved, and the searching efficiency cannot get prominent improvement. Although Koza has described using iteration to solve certain problems with GP [Koza92], compared with the program –cloning problem, all his GP applications are too simple. Accordingly, PCP's applications generally do not include *Loop* in their function set.

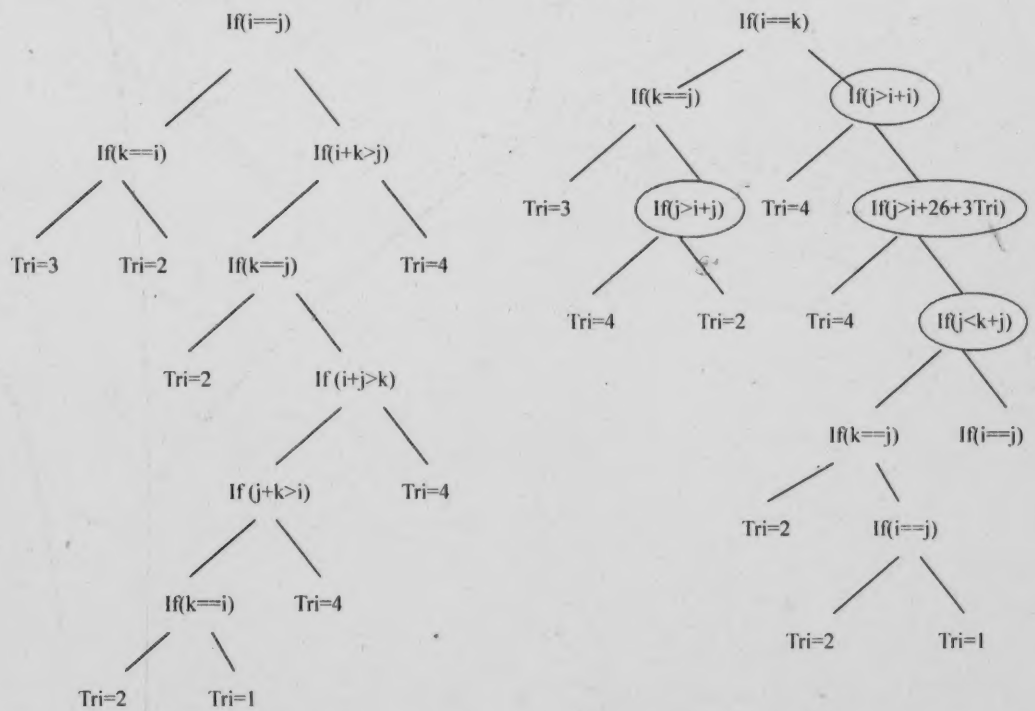
The elements in PCP's basic function set are described in Table 4.1.1, and triangle's program-cloning process used this basic function set.

+, - , *	Have two children, each of which can be +, -, *, /, %, *Constant, Input* and *Output*; children can be in any data type in arithmetic data types set, which includes *Integer, Float, Long* and *Double*, (the data typing measure manipulate the compatibility between them, see Section 3.2.1); and the return value is in the same type as (or compatible with) the children.
/, %	Include all constraints defined in the above column; and zero is allowed to be the second child, however whenever zero is encounter as the second child, an exception is automatically created by Java virtual machine and evaluation process on the current individual is terminated by assigning a poor score to that individual.

>, >=, <, <=, ==	Have two children of any mathematic data type; the return is a Boolean value; each of the children can be in node-type of +, -, *, /, %, *Constant, Input* and *Output*; and two constant values, or two same Input or Output variables are not allowed to appear on each side of a comparison express, e.g., *"3>=3"* and *"tri<tri"* are illegal (Note, it is hard and not excluded of the semantic insignificant conditions that two mathematic expresses with constant values appear on each side of a comparison expression, e.g., *"5+3-2*8>=3%10-9"*).
And, Or	Have two children with data type of Boolean; each can be ==, >, <, >=, <=, *And, Or* and *Not*; the return is a Boolean value; and must not be adopted by nodes in the second last layer in a parse tree because their children types do not include terminal type.
Not	Have one child with data type of Boolean; the child can be ==, >, <, >=, <=, *And, Or* and *Not*; the return is a Boolean value; and must not be adopted by nodes in the second last layer in a parse tree because their children types do not include terminal type.
If	Have three children; the first one has the data type of Boolean, and can be ==, >, <, >=, <=, *And, Or* and *Not*; the second and the third ones has no significant data type, and can be *Assignment, If, Seq,* and *Nop*; same as *And*, must not be adopted by nodes in the second last layer in a parse tree; and the second and third children should not be *Nop* simultaneously.
Seq	Have at least 2 and at most 5 children, and the number of children is determined randomly during GP execution; each children can be *Assignment, If* and *Seq*; *Seq* has no return value, and same as its children.
Assignment	Have two children of any mathematic data type; the first one can be *Input* and *Output*; and the second one can be *Input, Output, Constant,* +, -, *, / and %; the return value is in the same type as the children.

Table 4.1.1 Constraints of basic function-types

The third major step in preparing to use GP algorithm is to determine the fitness measurement. Fitness measurement is based on individual evaluation, which generally put individuals into a certain environment reflecting the characters of the target problem. Evaluation process executes the individual program within the environment and by comparing how close the execution result and the expectative result are, deduces the raw fitness, which directly corresponds to the difference. In the program-cloning problem, the evaluation environment is expressed by fitness cases, and in fact, the fitness cases are a series of test cases for the program being cloned. A delicately designed test cases set can well reveal the whole function contour of a program and at the same time, keep the size of itself to be as small as possible in order to achieve evaluation efficiency.



1) Best program from GP trials with 60 cases

2) Best program from GP trials with 30 cases

Figure 4.1.2 Comparing best programs from GP trials with 60 and 30 cases

For *Triangle.c*, our trials found that 60 is a suitable number of fitness cases. Note, although function testing for *Triangle.c* needs much fewer test cases than 60, the redundant cases help the GP's searching trend to be smoother and more stable. Evaluations with too few test cases risk guiding the

evolution develop toward premature directions. In our trials, with other approaches and parameters fixed, the GP searching with 60 cases, although with a reasonable slowing down on evolution speed, yields more functional results than that with 30 cases. Compared in Figure 4.1.2 are two simplified programs which are selected randomly from perfect solutions (meet all fitness cases) of GP searching with 60 cases and 30 cases. Watching Figure 4.1.2, we can see that the best program yielded by GP searching with 60 cases contains no insignificant sub-tree and misses only the first branch in *Triangle.c," if ((i<=0) || (j<=0) || (k<=0))"*. In contrast, the best program from GP searching with 30 cases not only misses the first branch in *Triangle.c*, but also contains four insignificant predications (as circled) and misses at least the predications of *(i+j<=k), (j+k <=i),* and *((i+k) <=j)*.

We also tested to increase the number of test cases to 90. However, with the evolving speed terribly getting down, the best adjust fitness for the 100th generation drops from 0.2 ~ 0.33 to around 0.067.

The individual evaluation process is described in Section 3.2.6. Based on the raw fitness, we carry out a series of measurements for the individual's final fitness, which is expected to reflect how well the individual fits the problem environment. As the basis of individual selection, the fitness drives the population to be more functional to solve the problem, as well, by adding diverse pressures into the fitness measurement, the fitness can drive the population to different directions, for example, to drive the individuals to have smaller size, or be more frugality [Koza92]. In the contrary, poor designed fitness measurement might misguide the population evolving to undesired directions. In our program cloning experiments, individual bloating [Ange98] [Banz02] [Lapo97] [Sofd96] is always a big problem due to the large number of primary elements. Aiming to individual bloating, we designed and successfully used the dynamic parsimony pressure method for fitness calculation during applying ECJ to program cloning (see Section 2.4). In PCP, we borrowed the fitness measurement of Koza's adjusted fitness [Koza92] and adopted the dynamic parsimony pressure method again. The concrete measurement is illustrated by the following formulas:

$$\textit{Adjusted Fitness} = \begin{cases} 1 / (1 + \textit{Standardized Fitness}); & \textit{when Individual Size} <= \textit{Size Limit} \\ 0 & \textit{otherwise} \end{cases} \qquad 1)$$

Standardized Fitness = Raw Fitness + Pressure; 2)

*Pressure = Individual Size * Pressure Coefficient [Individual Size / G];* 3)

Pressure Coefficient = {0.0f, 0.00001f, 0.00001f, 0.0001f, 0.0001f, 0.001f}; 4)

G = Length (Pressure Coefficient); 5)

*Size Limit = Length (Pressure Coefficient) * 40;* 6)

The element values in the array of *Pressure Coefficient* in formula 4) and the number *40* in formula 6) are experiment values.

As we described in Section 3.2.5, the local-optimization problem frequently occurs in our program-cloning applications. To overcome this problem, we adopt the dynamic fitness calculation method, which assigns higher score to individuals that satisfy fitness cases that have not been satisfied before. The calculation of the dynamic fitness is based on the adjusted fitness calculated through the formula 1) to 6).

The fourth step to prepare GP evolution is to set the environment parameters to control the run. In the problem of cloning *Triangle.c*, diverse values for the parameters are tried until the final values of parameters and settings are deemed mature and stable. They are listed as the following:

1) The population size is set around 3000 when all of the 16 primary elements in Table 4.1.1 are used in GP; and if we drop the number of element to 9, the population size can set around 1500.
2) The reasonable maximum generation is 300: in our trials (approximate 250 to 300 trials), most of the individuals that meet all fitness cases are produced within 200 generations with a very few cases created beyond 200 and within 250 generations.
3) Crossover probability is 90%.
4) Mutation probability is 40%. Although the generally mutation range is from 20% to 0.5%, we found 40% mutation probability works better than 20%, 10% and the even smaller ones. That is because we adopt redundant offspring (see Section 3.2.8), which introduces a strong convergent force in GP evolution, and higher mutation rate balances the too fast converging.
5) Number of fitness cases is 60;
6) Selection probability assigned to internal node within a parse trees is 90%;

7) The maximum depth of a tree produced during population initialization is 6;

8) The minimum depth of a tree produced during population initialization is 3;

9) The maximum depth of a tree produced by genetic operations is 9;

10) The range of random values is from 1 to 10;

11) Tournament size is 7;

12) Keep the best individual in the next generation; and

13) Root node is selectable for genetic operations.

The GP searching for clone of *Triangle.c* terminates after running 300 generations or when a perfect solution, which meets all fitness cases, is found.

The program-cloning results using PCP are much better than that we obtained using ECJ. With the population size being 3000, 7 out of 15 trials achieved perfect solutions and all of the perfect individuals were achieved within 200 generations. When fewer function types were adopted in GP searching, which contains only +, -, *, ==, >, <, *If, Seq* and *Assignment*, with population size being 1500, 9 out of 30 trials got perfect solutions. Recall the experiments using ECJ to cloning *Triangle.c*, a small number of function types, +, -, *, ==, >, < and *If*, were adopted, however only 3 out of 25 trials achieved perfect solutions, and each of them met only 31 fitness cases, which consequently are less functional than current solutions that met 60 cases.

4.2 Extract.cpp

As shown in Figure 2.5.2.1, *Extract.cpp* has only 20 lines of code. However, this program is actually more complex than *Triangle.c*. First, the maximum depth of branch embedding is 5. Second, more variables are adopted in this program: there are 5 independent input variables, *iDigit1, iDigit2, iDigit3, iDigit4* and *iDigit5*, two definitely defined output variables, location and hour, (i.e., *iExtractionLocation* and *iExtractionHour* in the code), and one result indicator (the standard output which displays the invalid code messages). Additionally, different from the condition in *Triangle.c* that the only output can be directly deduced from input values, one of extract's outputs is not only deduced from the five inputs but also affected by another output.

Fortunately, although in logic, the operations and calculations on the tree outputs are tangled together, from the point of view of implementation, we can separate the location's operations and hour's operations from the main body of program. Through the sub-function method as introduced in Secion 3.2.7, three GP searching processes were carried out consequently. In *Extract.cpp*, all of the sub-problems are pretty simply, and when one or more sub-problems is more complex, for example, replacing the "*iExtractionHour = (iDigit3 * iDigit5) - iDigit4*" with "*iExtractionHour = iDigit3 1/ iDigit2 + (iDigit3 * iDigit5) - iDigit4*", the advance of the sub-function method become more obvious.

Another character in applying GP searching to clone *Extract.cpp* is the usage of exception handling. In the original code of *extract.cp*, in case invalid digitals are input (e.g., smaller than 0 or causing unreasonable hour), a message is output to screen. For GP representation, we have to translate the display operation with a more manipulable operation. Although an output variable may function here, considering the purpose of the displaying, we can use the *Exp* to represent an undesired status and terminate the current execution immediately.

The first major step in preparing to use GP to clone the target program is to separate the whole problem into three cloning processes for three subroutines respectively. In PCP, three problem specification classes need to be set up: the first specification for hour and the second one for location can be defined simply as normal program specifications; and the third one, which represents the main body of the problem, must define a special variable, *strSubProblems*, to contain the class names of the first two specification classes.

The following major step in preparing to use genetic programming is to identify the set of terminals. All of the three specification classes specified the terminal node-types of *Input*, and instanced it with the five integer parameters for the code. Besides, the specification class for hour specified the *Output* and instanced it with the return variable of hour; the specification class for location specified the *Output* also and instanced it with the return variable of location. The specification class for the main body specified the terminal node-types of *Exp* and *SubFunction*, which was then instanced by the two variables, hour and location. All of the classes specified the terminal node-type of *Nop*.

All of the problem specification classes specified the terminal node-type of *Constant*. Not only from the code of *Extract.cpp*, but also from its specification, we could deduce the special constants: 0, 5, 17 and 23 as hour limits. We included these four numbers into the particular value array, *special*, for *constant* without introducing knowledge of the ready code into GP searching context. Then, the type typical values for programming (the second constant type described in Section 3.2.4) and random value generator are also adopted here. Considering that the hour, location and two outputs have the range from 0 to 23 and from 1 to 8 (see the specification in Section 2.5.2) respectively, we assign the range of random value generator from 0 to 9 for specification class of location and from 0 to 23 for classes of hour and main body.

The following major step in preparing to use genetic programming is to identify the set of functions. As shown by the original code, the operations on hour are limited to simple arithmetic calculations. Since this information should also be definitely depicted in problem specification, we can use it guiding us identify the corresponding function set. Without lose generality, its function set is {+, -, *, /, %, *Assign*}. The function set adopted in the other two classes is same as that used to clone the *Triangle.cpp*, i.e., {+, -, *, /, %, *Equal, GThan, LThan, GThanE, LThanE, And, Or, Not, If, Nop, Seq, Assign*}.

The fourth major step in preparing to use GP to clone the target program is to determine the fitness measurement. When cloning *Extract.cpp*, the fitness calculation formulas are same as that in the previous example. To evaluate the individuals, 50 fitness cases are defined for each of the three subroutine cloning. In our trials, the input values within the fitness cases are same for these three sub-problems. During individual evaluation process for the hour subroutine, the raw fitness scores a certain value corresponding to the absolute difference between the anticipating result and real result. In contrast, when measuring the raw fitness for the location routine, one mark is obtained when each cases is met by the current individual.

In the specification class for the main body, corresponding to the usage of *Exp*, an integer array variable is defined to express the status of execution. In this trial, there are only two execution statuses: 0 for success and 1 for fail. During evaluation, fail status is obtained when catching the exception and

otherwise success status is used. Since a general program generally provides several failing statuses, in order to express many statuses, we expanded the *Exp* to a function type that has one parameter representing the different failing conditions. Then, the elements in the integer array variable may have several statuses: 0 for success and others for fail, and through the exception message, the error codes are thrown from the individual calculation method and caught and used for fitness calculation by the evaluation method (see Section 3.2.2).

The next step to prepare GP evolution is to set the environment parameters to control the run. The GP searching process for each of the three subroutines adopts different parameters. The common settings are listed as bellow:

1) Number of fitness cases is 50;
2) Selection probability assigned to internal node within a parse trees is 90%;
3) The minimum depth of a tree produced during population initialization is 3;
4) Tournament size is 7;
5) Keep the best individual in the next generation; and
6) Root node is selectable for genetic operations.

The different parameters are listed in the Table 4.2.1

Parameter	Main body	Hour	Location
Population Size	1500	500	1500
Maximum Generation	150	50	50
Crossover Rate	90%	85%	85%
Mutation Rate	20%	10%	2%
Maximum depth of tree allowed in population initialization	7	6	6
Maximum depth of tree allowed during genetic operations	5	4	6
Range of random numbers	[0, 23]	[0, 9]	[0, 9]

Table 4.2.1 Different parameters for trials with *Extract.cpp*

To carry out GP searching for the three sub-problems, the entrance is GP initialization for cloning the main body subroutine. The special variable, *strSubProblems*, is then encountered, the sub-problems'

specification class names are obtained, and consequently, GP searching for the sub-problems is carried out one by one. After the two independent sub-problems are solved, GP searching for the main body cloning is carried out. Finally, best result from each of the independent cloning process is selected and fitted together to form the final solution for *Extract.cpp*.

When translating the problem from cloning the whole *Extract.cpp* to cloning three subroutines, we adopted the problem-specific knowledge that the calculations on hour and location depend only on the five parameters, which is obtained from the program specification but not the original code. By using sub-function method, *Extract.cpp*'s cloning process was greatly simplified and all of the three GP searching can easily find perfect solution. Three typical solutions obtained by each of the GP searching are listed as the following:

1) A solution for hour:

": = (Hour,-(+(-(D5,+(D4,Hour)),(D5,D3)),D5))"*, where *D3*, *D4* and *D5* are the last three parameters and for calculation the *Hour* has the constant value of 0, thus the solution can be simplified and expressed with *"Hour = D3*D5-D4"*, which is exactly identical with the target formula.

2) A solution for location:

"if(>(D1,D2),seq(seq(:=(Location,D2),:=(Location,D2),:=(Location,D2)),:=(Location,D2),:=(Location,D1)),if(>=(D1,D2),:=(Location,1),:=(Location,D2)))" where *D1* and *D2* are the first two parameters. The simplified solution can be expressed as:

```
if(D1>D2)
    Location = D1;
else if (D1 >= D2)
    Location = 1;
else
    Location = D2;
```

Although including redundant expression in the second predication, this solution is logically 100% correct.

3) A solution for the main body:

"if(>=(Hour,17),if(<(+(23,(0,0)),Hour),Exp,if(<(*(D4,17),Hour),Exp,==(D2,3))),if(<(Hour,0),Exp,if(<=(D1,0),Exp,if(>(Hour,5),Exp,Nop))))"* The simplified solution ca be expressed by Figure 4.2.1:

```
if(Hour >= 17)
    if(Hour>23)
        Exp
    Else if (Hour > D4+17)
        Exp
    Else
        D2=3
Else if(Hour<0)
    Exp
Else if(D1 <= 0)
    Exp
Else if(Hour > 5)
    Exp
```

Figure 4.2.1 Simplified solution in trials with *Extract.cpp*

As shown in *Extract.cpp* code, three invalid conditions are handled by the predications: *"iDigit1 * iDigit2 * iDigit3 * iDigit4 * iDigit5 == 0"*; *"iExtractionHour<0 || (iExtractionHour>5 && iExtractionHour<17) || iExtractionHour>23"* and *"((iDigit1+iDigit2) % 2) != 0"*. The above solution found the second predication, part of the first condition and none of the third one. The improvement of the result depends on the larger size of more uniform distribution of the fitness cases. However, using sub-function method, the current cloning trials are much more successful than that in our previously experiments with ECJ.

4.3 Journey.cpp

Same as the above two programs, the *Journey.cpp* cloning has also been experimented using ECJ. The original code is shown in Section 2.5.3. Although programmed within only 10 lines of code, *Journey.cpp* is difficult to be cloned not only because of the loop structure but also because of its

unusual solution space. As described in Section 4.1, even using very few primary elements and adopting execution time limitation, evaluations on *Loop* with randomly created predications may cost great computer time. However, when applying GP searching on program cloning, we generally adopted 21 basic node-types for solution construction. Hence, when *Loop* is added into primary element set and the GP searching is carried out on one normal personal computer: CPU speed 1.3GHz and memory 512M, it is not strange that our trials cannot achieve any high-quality result.

Consequently, same as the approach used in cloning *Extract.cpp*, we divide the whole problem into two sub-functions: one aims at searching the content inside the loop and the other one at searching the loop itself. As described in Section 2.5.3, the simplified searching process for the loop itself is easy to get a perfect result. However, when cloning the inside content, an interesting phenomenon, which relates to fitness-cases selection, is encountered and the trials reveals that fitness-cases used in GP seriously affect the success rate of GP searching. In addition, to determine the suitable number of test cases, the principle is not the more the better. The following description will focus on cloning the content inside loop.

The first major step in preparing to use genetic programming is to identify the set of functions. All of the basic function types listed in Table 4.1.1 are adopted. Then, the set of terminals should be determined. Here, the terminal types include *Output*, *Nop* and *Constant*. *Output* is instanced by variables: *n* and *peak*, and functions in this sample not only representing outputs but also representing inputs. To do so, *Output's* element *inits* is used to record the initial values for *Output* variables. For *Constant* type, special numbers: 1, 2 and 3, which are deduced from problem specification, are inserted into the particular value array. The fitness measurement is same as that in cloning *Triangle.cpp*. During evaluation of individuals, one hit is gained when one output of the individual result for each case achieves the expected value.

The environment parameters are set as following:

1) The population size is 3000.
2) The reasonable maximum generation is 300.
3) Crossover probability is 80%.

4) Mutation probability is 15%.

5) Number of fitness cases is 12, 15 and 40.

6) Selection probability assigned to internal node within a parse trees is 90%.

7) The maximum depth of a tree produced during population initialization is 6.

8) The minimum depth of a tree produced during population initialization is 3.

9) The maximum depth of a tree produced by genetic operations is 8.

10) The range of random values is from 1 to 10.

11) Tournament size is 7.

12) Keep the best individual in the next generation.

13) Root node is selectable for genetic operations.

When cloning the body content of *Journey.cpp*, the GP performance is not obviously sensitive to the crossover rate and mutation rate. With the crossover rate shifting among 80%, 85% and 90%, and mutation rate shifting among 10%, 15%, 20% and 30%, no improvement or deterioration is observed. In contrast, the setting of fitness cases affected the success rate of GP searching significantly.

When determine the fitness cases, we borrowed from software testing the decision table method in order to cover all conditions. As shown in Figure 2.5.3.1, the loop body of *Journey.cpp* has two branches, each of which has one predication. Thus the decision table can be expressed by the first two rows in Table 4.3.1.

n % 2 != 0	Ture	Ture	False	False
n > peak	Ture	False	Ture	False
40 cases averagely	10	10	10	10
12 cases averagely	3	3	3	3
12 cases not averagely	6	1	4	1
15 cases not averagely	8	1	5	1

Table 4.3.1 Decision table of *Journey.cpp*

In our first experiment, 40 fitness cases are adopted for individuals' evaluation and 10 cases for each condition, as shown in the third row in Table 4.3.1. Unfortunately, for such a simple piece of code, with the large population size of 3000, none of the 6 trials achieve hits more than 64 out of 80 cases

and the average hits are around 60. We deem the reason is in that with a big number of fitness cases but only four conditions, whenever a condition is satisfied by an individual, its fitness gets a big leap, thus there is short of consecutive improvement of the best individual and the evolution is easier to fall into local optimization points. Our following trials proved this conjecture.

In the following experiments, we improved the fitness cases by reducing the number of fitness cases to 12 and each condition has 3 cases. Then, in the three trials with the 12 fitness cases, the best individuals achieve 15, 16 and 15 out 24 cases, and the average fitness jump from around 0.05 to 0.11. Watching the solutions, we found that the simple expression, $n = n / 2$, is easy to be found, and in contrast, $n = 3 * n + 1$ and $peak = n$ are missed always. Then in the following trials, we redesign the fitness cases distribution as shown in the fifth row in Table 4.3.1, which gives more chances to the conditions with more complex operations. Then, in the three trials with the new fitness cases, the best individuals achieve 17, 18 and 22 out 24 cases, which shows obvious improvement from the previous experiment.

```
if( 3n%2>=1,
  seq(
    n=3n+1,
    if( 2p/N%7<=N,p=n)
  )
  seq(
    n=n/2,
    if(p<=n,p=n)
  )
)
```

Figure 4.3.1 The Best solution 1 in pseudo code

However, as we mentioned in Section 3.2.5, the fewer the cases, the less accurate the final solutions are. We then expand the number of fitness cases to 15, 2 out of 25 trials achieve perfect solutions, and the best one is shown as following:

if(>=(%((N,3),2),1),seq(:=(N,+(1,*(N,3))),if(<=(/(+(P,P),%(N,7)),N),:=(P,N),#)),seq(:=(N,/(N,2)),if(<=(P,N),:=(P,N),#)))*

Being simplified and expressed in pseudo code as shown in Figure 4.3.1, it obviously includes one bypass instruction as circled by the ellipse, then although meets all 15 cases, it fails to reflect the target problem. When we expand the number of fitness cases to 20, 1 out of 20 trials achieve perfect solution, and the best one is shown as following:

if(<(+(n,-(-(n,p),p)),+(n,n)),seq(if(!(>(%(/(n,1),%(9,7)),/(/(8,7),-(p,9))))),seq(:=(n,/(n,2)),if(>(p,n),#,:=(p,n))),:=(n,+((n,3),-(9,8)))),if(>(p,n),#,:=(p,n))),:=(n,-(p,1)))*

Being simplified and expressed in the following pseudo program in Figure 4.3.2, it obviously 100% identical with the problem.

```
if(n%2==0,
    n=n/2
    n=3n+1
)
if(p<=n)
    p=n
)
```

Figure 4.3.2 The Best solution 2 in pseudo code

Based on the experiments involved in cloning *Journey.cpp*, we can draw three conclusions: 1) loop structure is a very difficult point in program cloning, and direct cloning the whole loop structure is very unsuccessful. 2) the fitness case set is one of the key factors in program cloning, not only the number but also distribution affects the GP searching significantly, and assigning equal number of fitness cases to each conditions might not efficiently guide the GP searching. and 3) although for programs with different structures and different solution spaces, different parameters and settings are expected to use, there seems to be always a way for GP searching to work out the idea solution for that type of problem. We believe that with the constant accumulation of experiments with program cloning, rules for GP environment setting can be summarize to make most of the programs easy to be cloned.

4.4 NextDate

NextDate is come from Paul C. Jorgensen's [Jorg02], which is used to "illustrate a different kind of complexity – logical relationships among the input variables". *NextDate* a function of three variables: *month, date,* and *year*. It returns the date of the day after the input one, which is represented by the variable, *nextDate*. The variables of month, date and year have integer values subject to these conditions: 1<=m<=12, 1<=day<=31,1812<=year<=2012. Finally, leap year conditions are considered. With the leap year conditions, this problem is a little more complex than all of the previous problems, and a basic structured implementation of it includes about 50 lines of pseudo code [Jorg02].

Since program cloning need not and should not add any knowledge of the original program into GP searching environment, we omit the original program for this problem. Instead, assertions and restrictions deduced from the problem specification should be listed clearly in order to generate a set of efficient fitness cases. Same as *Extract.cpp*, we divide the whole problem into two parts: the main body checking the validation of the inputs values and a sub function making calculation of the next date, which is proved achieving much better results than cloning the whole program. Since the cloning process for the main body is similar to that in problems of *Extract.cpp* and *Commission*, we will focus on cloning the second part in our following description.

The first major step in preparing to use genetic programming is to identify the set of functions. All of the basic function types listed in Table 4.1.1 are adopted. Then, the set of terminals should be determined. Here, the terminal types include *Input*, *Output*, *Nop* and *Constant*. *Input* is instanced by the variables, *year, month* and *day* and *Output* is instanced by the variable, *nextDate*. For the *Constant* type, special numbers: 1,2,4,28,29,30,31,100 and 400, which are deduced from problem specification, are inserted into the special value array. The fitness measurement is same as that in cloning *Triangle.cpp*. When calculating the raw fitness of individuals, one mark is scored in case that the result for one of the outputs achieves the expected value for one fitness case.

For the evaluation of this problem, 100 fitness cases are designed. The designing of fitness cases include 5 steps: 1) list all restrictions for the problem, which, as shown in Table 4.4.1, includes 4 restrictions for *year*, three restrictions for *month* and two for *day*, 2) work out all the possible

conditions, which has 4*3*2=24 conditions, 3) select typical values or value ranges for the variables: *year, month* and *day* under each restrictions, 4) combine the typical values for 24 conditions, 5) according to the complexity of the calculation under each condition, assign different number of fitness cases to it, for example, the variable, *year*, is included by four restrictions, among which the proportion of cases is 1:2:3:4, thus with the total number of fitness cases 100, 10, 20 , 30 and 40 cases are assigned to the four conditions respectively.

Variable	Condition1	Condition2	Condition3	Condition4	Proportion of cases between different conditions
year	Normal year: 1999, 2002	Leap year exclude centennial: 2004, 1996	Centennial but no leap year: 2100, 1900	Centennial and leap year: 2000, 1600	1:2:3:4
month	February	Month with 31 days	Months with 30 days		2:4:4
day	Last day of a month	Other days			1:1

Table 4.4.1. Designing fitness cases

The main parameters that are different from other trials are set as following:

1) The population size is 3000.
2) The reasonable maximum generation is 250.
3) Crossover probability is 80%.
4) Mutation probability is 15%.
5) The maximum depth of a tree produced during population initialization is 7.
6) The minimum depth of a tree produced during population initialization is 2.
7) The maximum depth of a tree produced by genetic operations is 11.
8) The range of random values is from 1 to 12.

One of the typical perfect results is shown as the following:

seq(:=(N,D),:=(N,1),if(!(>=(*(-(9,D),+(0,5)),-(%(-(3,9),M),+(/(10,N),*(12,7))))),if(||(>(M,D),>(D,+(2,

(-(/(+(Y,Y),%(6,M)),Y),(D,Y))))),if(||(>(D,D),>(D,+(*(2,9),M))),#,:=(N,+(N,D))),:=(N,+(N,D))),:=(

N,+(N,D))))

Obviously, not only far from idea, the above program by far cannot reflect the problem. Although, for many cases that are outside the fitness cases set, like Feb. 28th, 2000, Feb. 29th, 2000 and Jan.1st, 2000, the above program works well, but for some, like, Jan. 31st, 2000, it fails. The key is that for such a solution, we don not know whether we can confide in it. And unfortunately, all of the perfect solutions, which are achieved with the parameters and settings described above, have the similar structure to this one.

From the content of the above solution, we can see that the un-success is greatly due to the unnecessary calculation between the variables, *year*, *month* and *day*. Although with integer values, mathematical calculation is not the main approach in this problem, and should be restricted between the input variables. One possible method is to define special node types, which although adopt the integer type, are available only for very simple calculations, for example, only calculations between a variable and a constant but not between variables are allowed. However, we did not experiment with this method, instead, we experience the definition for new data types in the following example.

4.5 Commission

Commission is also come from Paul C. Jorgensen's [Jorg02], it is "more typical of commercial computing and contains a mix of computation and decision making". This problem compute a rifle salesperson's commission monthly according to sales orders, each of which records how many locks, stocks and barrels he/she sales: locks cost $45, stocks cost $30, and barrels cost $25; 10% on sales up to $1000, 15% on the next $800, and 20% on any sale in excess of $1800. The most the salesperson could sell in a month is 70 locks, 80 stocks and 90 barrels [Jorg02]. Same as the previous example, no original code is referenced in the searching process preparation for the target program and fitness cases and other problem context are deduced directly from problem specification. However, in Jorgensen's, a typical structured implementation of it includes about 40 lines of pseudo-code.

As the most complex problem in our experiment, this problem can be divided into four sub-problems: 1) adding up the total number of locks, stocks and barrels respectively; 2) calculating

total sales amount; 3) calculating the salesperson's commission according to the total sales amount; and 4) checking the inputs and outputs to make sure the validation of the sale record for a month. The sub-problem 2) and 3) are obviously simple and are easy to achieve 100% correct programs. The sub-problem 1) presents a new condition in the program-cloning problem and is the emphasis in this section. The experiment with cloning sub-problem 4) will also be described here to explore the dilemma of efficiency and generality.

Firstly, let's look at the sub-problem 1), cumulating the total number for each product sold in one month. The inputs here are a series of sales orders, which, in high level programming language, can be represented with two dimension arrays. For example, {0, 2, 3} represents one sales order for two stocks and three barrels and {{0, 2, 3}, {1, 3, 4}} represents that two orders are gained in a month. The outputs for this problem are three total numbers for locks, stocks and barrels. For example, with the above two orders, the salesperson's achievement for that month is one lock, five stocks and seven barrels.

This problem exposes the problem of how to represent and carry out calculation on array. To solve this problem and as well as provide a possible solution for other combined data types, we introduced the approach of defining complex data types as terminal node-type.

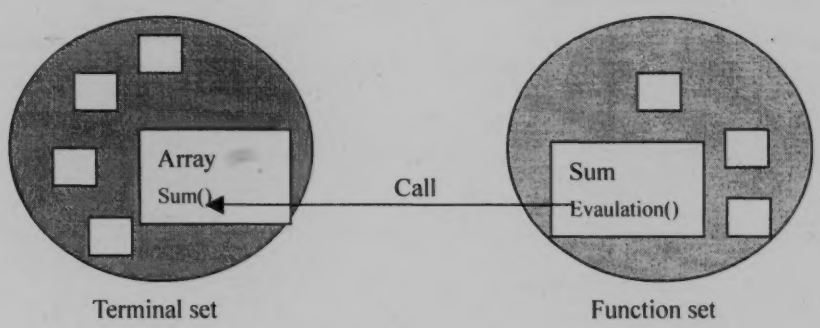


Figure 4.5.1 Adopting Array

As illustrated by Figure 4.5.1, a class, *Array*, is defined to extend the class, *Input*, which inherits the class of *Function* and represents all inputs variables as terminal node-types. Within *Array*, we can include any method defining possible operations and calculation for array structure. Accordingly, a

series of function node-types corresponding to *Array*'s methods are defined, their main children types are restricted to *Array* and their evaluation methods can call their matching methods in *Array*. For example, in this problem, we defined the methods, *Sum*, in *Array* to calculate the sum of elements in a one-dimension integer array or sum of elements at a specific position in a multi-dimension integer array; and then, a class of function node-type, *Sum*, is defined and from its evaluation method, calls the previous *Sum* method in *Array*. In GP, the usage of *Array* is same as that of *Input* and the usage of *Sum* is same as *Add*.

Therefore, the set of functions for cloning this sub-program include *Sum* and all of the basic function types listed in Table 4.1.1; and the set of terminals include *Array, Output, Nop* and *Constant*. Similar to *Input*, *Array* can represent all inputs of array type and needs to be instanced according to the specific problem. Here, it is instanced by the two-dimension array, *orders*, and the *Output* is instanced by the variable, *L, S* and *B*, to represent total numbers of locks, socks and barrels respectively. For the *Constant* type, special numbers: 0, 1,2, which serve for indexes of locks, socks and barrels in each order of *orders*, are inserted into the special value array. Then, in the problem specification class, 20 fitness cases are designed with the inputs structure in two-dimension arrays. When calculating the raw fitness of individuals, same measurement as that in cloning *Triangle.c* is adopted with one mark scored when the result of one of the outputs achieves the expected value for one case.

In the GP searching process, the main parameters adopted are listed as the following:

1) The population size is 500.
2) The reasonable maximum generation is 250.
3) Crossover probability is 80%.
4) Mutation probability is 15%.
5) The maximum depth of a tree produced during population initialization is 8.
6) The minimum depth of a tree produced during population initialization is 3.
7) The maximum depth of a tree produced by genetic operations is 5.
8) The range of random values is from 0 to 3.

In our trials, perfect solutions can be achieved easily. A typical one is shown as the following:

seq(:=(S,Sum(orders, 1)),:=(B,Sum(orders, 2)),:=(L,Sum(orders, 0)))

where *Sum(orders, i)* means the sum of elements at the second position in each array element of the two-dimension array. Obviously, the above solution is 100% correct in logic. According to our experiments, we believe that the approach for representing array and its operations is successful, and besides, it is easily to be expanded for other complex data structures or objects.

Secondly, let's look at the sub-problem 4), checking the inputs and outputs to ensure the validation of the sale record for a month. In our trials, similar parameter and setting values as that of cloning *Extract.cpp* are used, the number of fitness cases is 76 and special constant array includes the particular values deduced from specification. The results of our trials expose the problem of how to choose or make balance between efficiency and generality. Using the terminal set and function set listed in Table 4.1.1, we got one of the typical perfect solution as the following:

seq(if(>=(L,(*(*(B,L),5),S)),Exp,if(==(B,91),Exp,#)),if(==(B,91),Exp,if(==(S,81),Exp,if(==(L,71),Exp,#))))*

Although meets all of the fitness cases and many conditions of the problem, this solution is far from ideal. We believe that it is due to the unnecessary function set and terminal set for constructing individuals in order to achieve generality of the program-cloning process with GP.

By taking off the arithmetic operations, +, -, *, / and %, from function set, we get a typical solution as the following:

if(&&(>(L,0),&&(>(S,0),&&(<=(L,70),<(S,81)))),if(&&(<(S,81),>=(B,1)),if(&&(<=(B,S),<(L,0)),#,if(<=(B,90),:=(B,S),Exp)),Exp),Exp)

Although not very obviously, the above solution is 100% correct in logic, with the cost of sacrificing the generality of cloning process. We believe that for different applications, the program-cloning process focuses on different targets and has different limits, and how to balance between efficiency and generality should be determined by the concrete application. We will further discuss this point in the Chapter 5, Applying program cloning in software engineering.

Chapter 5

Applying program cloning in software engineering

Software development is in a state of crisis, and the industry as a whole is in a state of denial. The software industry has an abysmal record of bringing software projects in success. Research from The Standish Group* shows a staggering 31.1% of projects will be canceled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. According the Standish Group, the United State spends more than $250 billion each year on IT application development of approximately 175,000 projects. Of these 31% are cancelled, 53% are changed, 16% successful. The average cost of a development project for a large company is $2,322,000; for a medium company, it is $1,331,000; and for a small company, it is $434,000. These figures paint a black picture and clearly our ideas on software production processes are failing to deliver the required results. Hence, the topic requires a radical shift in its approaches! This chapter presents some initial investigations into a new direction in supporting alternative production process – software cloning of source code.

Although the generation and application of software clones remains unexplored, it is believed that this is a fundamental technology that can have many different applications within a software engineering environment. Unfortunately, given that we are still at the "proof of concept" stage, it is impossible to exactly predict where the technology will succeed and where it will fail; equally it is impossible to give detailed descriptions of the potential applications as these will require significant research programs of their own to establish their details and results. In fact, we believe that exploring these areas and techniques will present significant challenges and that several decades of effort may be

---

* The Standish Group. CHOAS Chronicles II, The Standish Group International Inc., 2001

required to establish results that are applicable within an industrial context. Hence, the following should be considered as brief initial thoughts on how to apply this technique rather than a polished statement of application; in additional, it provides some insight into the motivations for conducting this work [Recm03].

While it is attractive to utilize the program-cloning techniques to automatically create programs for general purposes in high-level languages, our trials show that it is a long way of applying program cloning to produce programs matching hand-written codes. However, we believe, the program-cloning technique can practically benefit multiple topics in software engineering domain. Six of the possible domains are studied and described in this chapter:

1) Complexity measurement,
2) Mutant software test,
3) N-Version software design,
4) Test first,
5) Test data evaluation, and
6) Automatic test data generation

In the following six sections, we will discuss how to utilize the program-cloning technique to solve problems in these domains respectively. Each section includes an introduction of the target domain, its current solutions and existing drawbacks or weaknesses and analysis of how the program-cloning technique manipulates the problem, overtake or bypass current solutions' weaknesses.

5.1 Complexity measurement

Software Engineering has struggled for a long time to derive a comprehensive idea of complexity. From the early work of McCabe [Mcca76] and Halstead [Hals77] to the present time, researchers have struggled to define the idea of complexity. As with this early work, most theories of software complexity seek to measure structural properties of an arbitrary system and hence the measure of complexity becomes highly correlated with the programming style of the production team. Hence, the same problem (or specification) when implemented by different programming teams will in general

produce significantly different ideas (and values) of complexity. Therefore, we can think of these complexity measures as attempting to encode the complexity of the implemented solution rather than that of the problem statement.

Following work in the "science of complexity" [Gell95], believes that several different ideas of complexity exist for software, including but not limited to ideas of structural complexity. What is required are many different measures of complexity - capturing all of our intuitive ideas of its entire definition. Hence, our *interest* here is not with structural complexity but with *system complexity*. We want to consider the system as a black-box, see only the function specification and ask: how complex is this system (or box)? Clearly, as the system is a black-box, we can only see the external interactions of the system and hence these will become the sole drivers of our complexity expression. In addition, as we are dealing with a black-box, our expression will be independent of the implementation; and in fact will be independent of the implementation technology. Our viewpoint or model of this facet of complexity is: how difficult is it to arrive at a complete understanding of the system? Is system A more difficult to understand than system B?

This orientation seems to be echoed by many recent directions and ideas from generic complexity descriptions of arbitrary artifacts [Pine88], where properties of the entire system are expressed as an expression with regard to the system's boundary. Clearly, we could attempt to measure this concept by an individual (or a group of individuals) undertaking the task of extracting the meaning of an arbitrary system and measuring their performance or difficulty in achieving their objective. While this approach is possible, it is in general extremely costing in terms and time and money. Hence, what is required is a secondary measure or model of this activity, which alleviates these problems.

It is believed that program cloning represents such a secondary model; again we are trying to learn or understand the behavior of the system, but this time an automated process is undertaking the task rather than a human being. As above, our *measure of complexity* is the difficulty experienced in this learning process; and in the cloning system this is represented in two dimensions:

1) The percentage of the sampled input space correctly modelled. That is: is the clone able to derive the same output as the original system given the same input? Hopefully, over time this will reach

one hundred percent, moving us into our second dimension.

2) The effort of the learning process for perfectly cloning the target software. This can be simply re-expressed as the number of behavioral examples required by the cloning process to reach this point, for example, in GP, the number of population, maximum generations allowed, the number of fitness cases and of sub-processes.

Our trials have demonstrated this effects that programs with different complexity require different effort to achieve a perfect solution. From the *Triangle.c* to *Commission*, it is obvious that the target problems being cloned become more and more complex: from a single output to multiple outputs, from pure branch structures with simple calculation to mixture of validation checking, branch structure and mathematical calculation, and from basic data type to the array data structure. Consequently, the GP searching process had to put on more efforts when finding the perfect solutions:

1) An obvious trend is that suitable fitness cases become more difficult to be determined. In order to provide suitable evolving environment for GP searching, the fitness cases must not only reflect every character of the problem but also emphasize complex conditions or special usages by more cases. In our cloning practice, we have tried using the decision-table method (see Section 4.3), giving more cases to conditions under which complex operations are carried out (see Section 4.4), and restricting the case number for each normal condition to avoid local optimization (see Section 4.5). However, when applying the program-cloning method to automatically express the entire system for complexity measurement, none of the above approaches should be adopted for cases determination. Instead, most of the fitness cases should be created automatically, e.g., created randomly, hence, the more complex the problem, the harder to locate efficient fitness cases and the more random cases should be created. Hence, it is reasonable to include the number of fitness cases as one measurement parameter for problem complexity.

2) Further, in order to locate a reasonable solution with limited resources, the whole problem has to be cut into more sub-problems to be cloned. For example, when cloning *Triangle.c*, no sub-problem division is adopted and the cloning process for the whole problem achieves reasonable results (see Figure 4.1.2); and when cloning *Extract.cpp*, we divided the entire

problem into three parts and cloned them sequentially. This division greatly simplifies the cloning process by avoiding coupling of variables or operations from the sub-problems and reducing the primary element set for each sub-problem. Therefore, in order to achieve perfect solutions with identical resources, e.g., GP's population number or running time, the more complex the target problem, the more sub-routines should the problem be divided into during the program-cloning process.

3) Under certain conditions, special data structures are necessary to be added into the primary elements to construct solutions. Both program structures and data structures are analyzed in previous complexity measurements. Although different problems require different primary elements for solution construction, our trials, in order to achieve generality, adopted a series of identical primary elements (see Table 4.1.1) for each program-cloning applications. Nevertheless, for certain problems, shortage of certain elements, e.g., particular data structures or functions, may result in practical impossibility of program cloning's success. Selection of data structures, terminal nodes and functions can be carried out manually according to the problem specification or the interface specification. Although the manual selection decreases the automation slightly, it improves the cloning efficiency significantly. Accordingly, a complexity weight corresponding to the particular data structure should be considered in the problem's complexity measurement.

In fact, not only data structures, but also different terminal nodes and functions can express different information and thus have different levels of complexity. For example, the assignment operation is obviously simpler than If-then-else structure; the Loop is generally more powerful than If-then-else; and array data structure is more complex than basic data types. However, when assigning different complexity weights to different primary elements, we should also consider the evaluation time generally required by different elements. When measuring a problem's complexity, we consider the total resource (include time) consumed for cloning it, hence when determine an element's complex weight, the more time-consuming an element, the more reduction the complex weight gets. For example, the Loop structure is a little more complex than If-then-else structure, however the evaluation on it costs much more time than any other function,

thus it should has a relative low complex weight in order to avoid assigning too high complexity to the loop using problems, according to the great amount of computer resource consumed on them.

Having no practical results applying program cloning to calculate complexity, we do not intend to provide a detailed measurement method in this thesis. However, the two dimensions of using our cloning process to measure complexity are supported by and deduced from our trials, and can serve as basis for further work in this direction.

5.2 Mutant software test

Software Mutants are simply new versions of a system which possess a deviation from the original. To date, applications or techniques using mutants have received limited attention. Traditionally, mutants have been used in mutation testing [Dels79] and have recently begun to become integrated into commercial testing tools and approaches [Corp96]. In addition, they are starting to find new roles within Software Engineering; for example, Briand et al. [Brlw02] used them as an input mechanism to drive exploratory simulations. Traditionally, mutation testing is based upon seeding the original program with a fault by applying a mutation operator, such as changing an addition operator to a subtraction operator. We then ask the question: can the test set differentiate the mutated program from the original? Our basic premise is: given an appropriate set of mutation operators, if a test set differentiates all the mutants generated by these operators, then since it can differentiate these planned imperfections, it should also be good at differentiating other unplanned imperfections, i.e. faults.

The principal reason for their limited application is that although mutation is a powerful and very general technique, a number of problems are associated with its traditional implementation approach. For example, the standard set of mutant operators often leads to a vast number of mutants being produced; while some progress has been made on this problem [Mrbo99][Oflr96], the issue remains unresolved. In addition, this mutant operator approach can cause mutants which have infeasible paths within their formulation making them difficult and sometimes impossible to explore [Ofpa97]. Finally, traditional generation approaches often produce mutants that are syntactically different but

semantically identical. Clearly, this causes problems in applications, such as mutation testing, as these mutants will never be differentiated from the original system. Again, progress has been made in detecting and removing these semantic equivalent mutants from the production process [Hahd00][Hihd99] – but further advances are required to enable the technique to surpass this problem.

One of the advantages of using genetic programming as a mutant production mechanism is that we are able to abandon the traditional generational approach. For example, Emer and Vergilio adopted Chameleon, a generic genetic programming tool [Emve03][Spin01], to produce alternative programs (mutants). Different from Chameleon (as described in Section 1.2), Program cloning focuses on producing programs which are generic in function as well as in grammar. Hence, we believe that our program-cloning method is adoptable to create mutants for general-purpose software. To apply program-cloning method to mutation testing, we conceive three approaches:

Firstly, the programming-cloning method can be expanded to a program decomposition and recomposition process. As revealed by our trials, the more correlative the primary components are to the target problem, the more efficient the GP searching is and the higher quality the final solutions are. By adding a little program analyzer, which simply abstracts primary elements from original programs being mutated, into the pre-process of program cloning, we automatically force the cloning process to become more problem specific. Consequently, the programming-cloning procedure turns into pure program understanding or problem learning process, and this process learns the behavior of the original program from its external interfaces, fitness cases, and gains its composing elements from the program decomposition procedure. A perfect solution of program cloning is the result of the learning process. In the same way as human beings, mutant production is based on an understanding of the problem -- the perfect solutions. Mutants can be individuals achieved from mutation operations on perfect solutions, parents of perfect solutions, or any other imperfect solution, as long as they omit certain fitness cases. Mutants created this way are guaranteed to be different from the original program because at least one external behavioral pattern functions differently in the mutants and the original program.

Secondly, by choosing individuals with different evaluation values, we can produce a spectrum of

mutants which differ by varying amounts from the original.

Finally, the possibility also exists for applying genetic programming mechanisms with samples of the original programs behavioral patterns which themselves have been mutated to introduce further mechanisms to differentiate the original from its mutants.

5.3 N-Version software design

Today there exist many systems where failure or malfunction can lead to disastrous situations potentially resulting in the loss of human life. Computers and their associated software are increasingly being used to control critical operations in many fields. Examples can be found in the aerospace, energy, medical and defense industries. The software controlling these systems must be reliable and safe. Unfortunately, current practices fail to guarantee such qualities. All current approaches produce software which has a number of faults contained within it. To counteract these deficiencies practitioners employ software fault tolerance techniques. These techniques work by providing additional copies of the functional components, these copies are functionally equivalent; but are derived by a different process from the original. It is hoped that the copies will not contain the same faults as the original and hence regardless of the situation at least one copy should produce the correct solution.

Traditional fault tolerance techniques fall into two categories: design and data diversity. Data diversity techniques first appeared in an excellent paper by Amman and Knight [Amkn88]. Unfortunately these techniques have failed to progress from this starting point. On the other hand design diversity techniques have received a great deal of attention [Lyhe93][Torr00]. Both fault tolerance strategies can be subdivided into two sub areas: static and dynamic.

The most common manifestation of static fault tolerance is N-version programming. From an initial specification, N functionally equivalent implementations are derived, usually by different programming teams. These N versions are subsequently executed in parallel, with each version receiving identical inputs. At various points during the execution, the outputs of the N versions are compared and a decision mechanism decides by consensus what the *correct* output is given the N

opinions. The decision mechanism subsequently sends a status indicator to each version after comparison of the results. Typical indicators are *continue*, *continue after altering output to the consensus result* or *terminate*.

Dynamic fault tolerance also works by supplying redundant components. The main difference is that these components only come into action after an error has been found. Typically this technique has four phases of operation: error detection, damage confinement and assessment, error recovery and fault treatment and continued service. Perhaps the best known and simplest dynamic fault tolerance scheme is the recovery block. Recovery blocks are traditional blocks as seen in most programming languages except that the entrance is an automatic recovery point and the exit an acceptance test. The acceptance test estimates if the block's calculations have resulted in an acceptable output. If so, the block finishes. If not, the block is retried with an alternative module (sometimes offering a degraded service) attempting the computation. This is repeated until a suitable output is found or no more alternative blocks remain. Each technique has its advantages and disadvantages and they can also be considered as complementary. Both techniques have their analogues in the data diversity area.

All of these approaches suffer from a common issue – that of producing a clone (or functional equivalent) or clones of the original system or sub systems. Clearly, if undertaken manually this has massive cost implications. Also, the manual approach suffers from the fact that it is common for programmers to produce the same solution to the problem, thus negating any advantage from the approach [Knle86]. (Commonly referred to as the "independence assumption".) This problem is further exacerbated - if the programmers involved in producing the alternative version are the producer's of the original version. (The same type of problem can be found in the production of other pieces of software, such as exception handlers and unit testing code.) If these techniques are to progress and become mainstream production approaches, these two issues need to be resolved. Unfortunately, it is difficult to see that any possibility exists to resolve these problems via traditional human-oriented production means and mechanisms. In order to resolve these problems, the alternative version or versions need to be produced automatically, or at least semi-automatically, which, we believe, can be implemented by our program-cloning method.

With the purpose of simulating the independent design process for additional versions of a problem, we assume that the precondition of program cloning is the detailed problem specification. Problem knowledge is deduced from the specification, and according to it the functional set and terminal set are determined. The fitness cases, which should reflect the full functional behaviors of the target block being N-version designed, are also deduced from the problem specification. The same functional set and terminal set, which directly echo the problem specific knowledge, can be adopted when cloning different versions of the program. However, the fitness cases, which are volatile, should be independently deduced from the function specification for cloning each version of the program. To create additional versions for complex problems, dividing the whole problem into several sub-functions is generally helpful in order to achieve high quality solutions. Although the problem dividing operation is carried out manually, our trials revealed that the little sacrifice on automaticity benefit the program-cloning process significantly, and moreover, computer aided tools can alleviate the manual load. It is believed that to create a program's different versions, different problem-dividing processes are desired. The more formal the problem specification, the easier the program-cloning process combines with the computer aided tools.

5.4 Test first

In recent years, an alternative approach to software production has emerged – agile (or lightweight) methodologies. While several approaches exist under this banner, the movement has been dominated by the Extreme Programming (XP) approach [Fowl00]. XP consists of about a dozen practices which are integrated together to produce the new methodology. These practices place a strong emphasis on testing and XP requires that testing becomes the foundation of the development process [Beck99] with every programmer writing tests as a precursor to writing their production code [Beck02].

The approach to production again creates a possibility for interaction with the genetic programming system. By producing tests for the system, the programmer is producing a mechanism to allow us to view and interact with the external behavior of the system. In XP the tests effectively act as the specification of the system and are clearly executable; hence the genetic programming system has

the opportunity to automatically produce the system given the test code. Clearly, the code is unlikely to possess the same *"quality characteristics"* as code produced by a human programmer, but stylistic issues have less importance in this initial version (of the system) within an XP environment. In XP, the role of the initial version is simply to successfully pass the test case; subsequent to passing, the code will be refactored by a programmer to produce a final (*high quality*) version (or at least an initial final version – as further refactoring is likely) of the system. Alternatively, the code may not be used directly, but could simply be supplied to the programmer as a mental assistance in solving the production problem.

5.5 Test data evaluation

A question related to the testing activity, is to know whether a program has been tested enough or when to stop testing [Rawe85] [Emve03]. One direction is to adopt certain models to evaluate the efficiency of the current testing process. For example, Dalal and Mallows [Dama88], present a strategy for successive determination, based on the distribution of the fault-finding rate deduced from severity of bugs and the time interval to find them. Another direction is test data evaluation. Related techniques include statistical testing [Mill72] and mutation analysis [Demi78]. Statistical testing estimates the remaining errors in the software under test (SUT) by seeding errors into it, applying test data to it, and calculating estimate errors based on the number of the seeded/unseeded errors discovered. Mutant analysis is based on mutation testing and rests on two assumptions that the SUT is almost correct and tests that uncover simple errors can also uncover deeper errors.

One problem with mutation analysis is the expensive cost on creating mutants for the SUT, and in the second section of this chapter, we have provided a solution of using the program-cloning method to automatically create mutants. However two other problems exist in the above statistical testing. Firstly, suitable error seeds, each of which focuses on the problem's different aspects and aims to different problems that may be exist in the testing process, are difficult to be located, and the traditional manual approach is time costing. Secondly, as the direct product of human intelligence, software may include complex behavior, in contrast, statistical methods are always inclined to

simplify the target problems, thus different models used to measure the testing efficiencies can make dramatically different results.

As a way bypassing the above problems, the program-cloning method provides a new approach for test-data evaluation and consequently provides conditions for making decisions to stop or continue the current software testing process. Our method is illustrated by Figure 5.5.1.

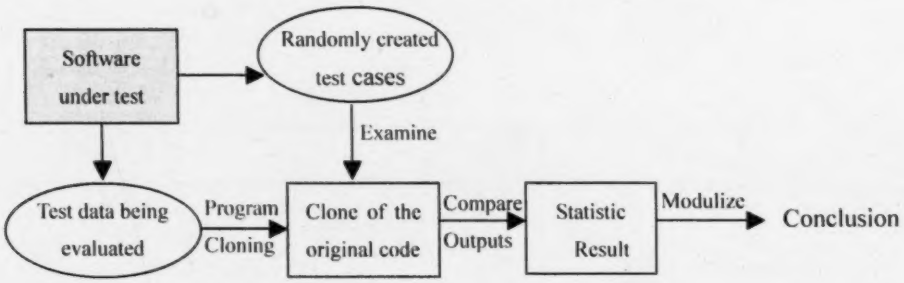


Figure 5.5.1 Test data evaluation

As shown by the figure, we assume two preconditions: the software under test (SUT) and the test data being evaluated. Through program cloning under the environment of the test data, a clone of the original code is produced, and its quality, revealed by our trials, mainly depends on the fitness cases (the test cases being evaluated). Thus the more mature and efficient the test data, the higher quality the clone is. In the following step, a great number of random test cases are created and evaluated by both the original program and the clone; and by comparing their outputs, we measure the quality of the clone, according to which the quality of the test data being evaluated is deduced.

For example, the SUT is *Triangle.c* (see Figure 2.3.1), and the test data being evaluated are:

i: *{30,38,25, 49,47,82,18, **0**,49,40,26,67,75,54,50,-80,75,88,59, 57,40,84,45, 96,26,69,46, 97,59,27, 50,5, 61,3,60,-39,14,72,95, 0,43,95,74, 39,75,52,17,176,93,72,13, 32,8, 47,40,90,52,47,15,4},*

j: *{72,67,25, 95,37,82,18,-34,56,40,26,28,19,28,50, 34,97,88,59,130,44,84,45,192,27,22,46,111,86,93, 50,96, 82,2,60,-29,67,66,95,94,76,36,74,108,83,52,17,86,67, 79,13,131,35,69,40,88,29,86,15, **0**},*

k: *{49,67,25,161,38,30,18,-85,22,65,26,32,93,54,50, 44,49,47,59, 56,80,12,45, -19,22,69,46, 3,67,93, 50,111,63,3,60, 3, 61,72,95,75,58,95,74, 67,71,63,17,73,46, 72,13, **-2**,41,69,40,181,65,86,15,99},*

tri: *{ 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1,2,3, 4, 1,2, 3,4, 1, 2, 3, 4,1, 2, 3,4, 1, 2, 3, **4**, 1, 2, 3, 4, 1, 2, 3, 4}*

Note that one condition missed by the above test data is that when ***k*** equals 0, ***tri*** is 4; so, generally, an ideal clone program created according to the defective test data set is an mutant of the original one, which can be expressed with the form:

triangle body 1 + *"if ((i<0) || (j<=0) | |(k<=-2))"*+ *triangle body 3*, where the *triangle body 1* includes *Triangle.c*'s first 2 lines of code, *triangle body 3* includes *Triangle.c*'s lines of code after the third line, and *"if ((i<0) || (j<=0) | |(k<=-2))"* is a mutant of *Triangle.c*'s third line, *"if ((i<0) || (j<=0) | |(k<=0))"*. In the following step, a series of random test cases are created by *Triangle.c*, and they are:

i: *{-11, 0, 21, 21, 21, 91, 31, 11, 9, 31, 61, 31, 61, 51, 51, 31, 61, 51, 81, 31, 11, 31, 11, 72, 71, 31, 41, 31, 41, 91, 82},*

j: *{ 21, 91,-11, 0, 31, 81, 31, 11, 9, 31, 61, 51, 51, 31, 61, 41, 41, 31, 61, 41, 31, 72, 71, 31, 11, 31, 41, 91, 82, 31, 41},*

k: *{ 31, 21, 31,-11,-11, **0**, 31, 11, 9, 51, 51, 31, 61, 31, 61, 51, 81, 41, 41, 72, 71, 41, 31, 41, 31, 91, 82, 31, 41, 31, 41},*

tri: *{ 4, 4, 4, 4, 4, 4, 3, 3, 3, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}*

When we applying these test cases on the clone program, the sixth case creates an output 1, which is different from the standard result 4. This different is detected by the comparing procedure, and the test data under evaluation are proved to be inefficient.

5.6 Automatic test data generation

Another application for the program-cloning method is automatic test data generation. Test data generation is the most important and difficult part in software testing domain [Ould91]. In recent years, automatically providing test data by genetic algorithms, such as GA, simulation annealing and taboo, seems to be a hopeful direction, and many researches have got achievements in this domain [Mims01][Trac00][Watk95]. GP, as an efficient machine learning approach, has also been tried in software testing [Emve03], yet to date, none has successfully applied GP to automatically generate test

data. The problem is: 1) when the genetic algorithms or other methods are adopted to automatically create test data, the expected outputs of the algorithms are a set of test cases for the software under test; and 2) the output solutions produced by GP algorithm are executable programs, which results in the difficulty of representing and providing test data directly.

Using the program-cloning method, we can produce a solution that overcomes the above problems. In our method, the output of GP's automatic test data generator is not the test cases, but a program, acting as a test-data generation. This test data generator works according to the logic in the problem specification, or in other words, the created test data generator includes all function logic of the target problem, according to which, the test data are produced. This idea comes from inspiration by the manual process for test-data designing and the symbolic-execution method for software testing. In both approaches, the software under test or the target problem are studied and understood in either intelligent or mechanical way; and consequently, the test data are designed based on the understanding of the problem or program. Correspondingly, in our method of cloning the test data generator, the software under test (SUT) is learned and understood by the GP algorithm, and then, a test data generation mechanism, the created test data generator, is established based on the understanding of the problem.

The main idea of this method is described as the following: A test data generator (TDG) is a program which produces test data for a particular SUT (software under test) according to its internal logic. The internal logic, in order to work along with the TDG, is hard coded into TDG's program. Conventionally, people can create such a TDG program manually and include the SUT's logic based on personal understanding of the target problem. A test data generator in this style is very efficient, but also very costly and may have similar logic errors as that in the SUT. Fortunately, with program cloning, a program, instead of human, can understand the target problem automatically and then create TDGs.

In the concrete implementation of this method, each individual in GP can be wrapped in a particular routine frame to form a TDG, or each individual can represent a series of components instancing the routine frame. For example, the test data generator for *Triangle.c* in Figure 5.6.1 is a

manually created program, its lines of code in plain text compose the framework to construct TDG and the lines of code in italic and bold are contents being searched with the program-cloning method. The objective function for each individual is the statement/branch coverage, and this objective function can be achieved by dynamically running the SUT using test cases generated by this TDG individual. Suitable terminal/function types for constructing the TDG can be deduced by analyzing the SUT. The whole method is illustrated by Figure 5.6.2.

```
package apgp;
import apgp.util.*;

public class CreateTriangleTestCases{
    public static void main(String[] args){
        if(args.length<1){
            System.out.println("java CreateTriangleTestCases n");
            System.exit(0);
        }
        MersenneTwisterFast random = new MersenneTwisterFast((int)System.currentTimeMillis());
        int n = new Integer(args[0]).intValue();
        int[][] inputs = new int[3][n];
        int[] outputs = new int[n];
        float fr = 0;
        for(int i=0;i<n;i++){
            outputs[i] = i%4 + 1;
            int a=0,b=0,c=0;
            switch(outputs[i]){
            case 1:
                do{
                    a = random.nextInt(100)+1;
                    b = random.nextInt(100)+1;
                    c = random.nextInt(100)+1;
                }while(a+b <= c || a+c <= b || c+b <= a);
                inputs[0][i] = a;
                inputs[1][i] = b;
                inputs[2][i] = c;
                break;
            case 2:
                do{
                    a = random.nextInt(100)+1;
```

```
            b = random.nextInt(100)+1;
        }while(a+a <= b || b == a);
        fr = random.nextFloat();
        if(fr<0.3333f){
            inputs[0][i] = a;
            inputs[1][i] = a;
            inputs[2][i] = b;
        }else if(fr<0.6667f){
            inputs[0][i] = b;
            inputs[1][i] = a;
            inputs[2][i] = a;
        }else{
            inputs[0][i] = a;
            inputs[1][i] = b;
            inputs[2][i] = a;
        }
        break;
    case 3:
        a = random.nextInt(100)+1;
        inputs[0][i] = a;
        inputs[1][i] = a;
        inputs[2][i] = a;
        break;
    case 4:
        if(i%7==0){       // <= 0
            a = - random.nextInt(100);
            b = random.nextInt(200) - 100;
            c = random.nextInt(200) - 100;
        }
        else{ // a+b <= c
            b = random.nextInt(100)+1;
            c = random.nextInt(100)+1;
            a = c+b+random.nextInt(20);
        }
        fr = random.nextFloat();
        if(fr<0.3333f){
            inputs[0][i] = a;
            inputs[1][i] = b;
            inputs[2][i] = c;
        }
        else if(fr<0.6667f){
```

```
                    inputs[0][i] = c;
                    inputs[1][i] = b;
                    inputs[2][i] = a;
                }else{
                    inputs[0][i] = b;
                    inputs[1][i] = a;
                    inputs[2][i] = c;
                }
            }
        }
        //present int[][] inputs, int[] outputs
        StringBuffer sb = new StringBuffer();
        for(int i=0;i<inputs.length;i++){
            sb.append("{");
            for(int j=0;j<inputs[0].length;j++){
                sb.append(inputs[i][j]);
                if(j != inputs[0].length - 1)
                    sb.append(',');
                else
                    sb.append('}');
            }
            if(i != inputs.length - 1)
                sb.append(',');
            else
                sb.append(" };\n");
        }
        sb.append("{");
        for(int i=0;i<outputs.length;i++){
            sb.append(outputs[i]);
            if(i != outputs.length - 1)
                sb.append(',');
            else
                sb.append("};");
        }
        System.out.println(sb);
    }
}
```

Figure 5.6.1 Hand-written test data generator for *Triangle.c*

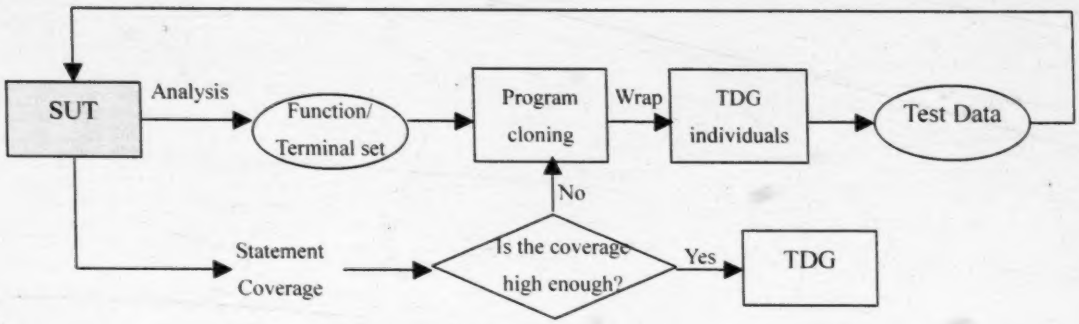


Figure 5.6.2 Automatic test data generation

While our trials focus on the program-cloning technique itself, all of the possible applications are only conceived by theory. Nevertheless, we believe in that the program-cloning method may benefit these domains as we described.

Chapter 6

Conclusions and Future work

6.1 Conclusion

In this thesis, we have described our experiments with program cloning and analyzed its potential applications in software engineering. From what has been discussed in the previous chapters, we can conclude that program cloning, as an application of GP, is a theoretically and practically sound approach for solving software engineering problems.

From the point of view of GP's application, the basic concept of program cloning (defined in Section 1.2) directly expands GP's ability of automatic creating solutions from restricted problems to arbitrary problems as long as they can be solved using a program. Our samples have shown its possibility and practicability by simulating programs for diverse problems. These sample problems, which have different levels of complexity and can be reflected by inputs/outputs profiles, are very general. Different from the conventional problems created by GP, they can be expressed in any high-level computer language and serve a wide variety of purposes, i.e., the programs that program cloning create are more general and have greater similarity to those produced by programmers.

Besides the long-term purpose of using program cloning to replace part of programmers' work on general-purpose coding, the program-cloning technique can also benefit a number of domains in software engineering. In Chapter 5, we have listed 6 possible applications: complexity measurement, software mutant creation, N-version design, test first, test data evaluation and automatic test data generation. However, we believe that along with the developing of the program-cloning technique, more applications may be discovered. Thus, program cloning is not only a GP application but also an extension of GP and an approach for software engineering.

As an application of GP, program cloning does not introduce a new algorithm to automatically

produce programs. However, one important contribution of this work is the practical experiments in translating the potential possibility that GP can be adopted to create general-purpose programs into reality. As well, a trial version of a program-cloning specific GP package was implemented, and during the implementation process, practical methods which fit GP into the program-cloning problem, such as exception handling, sub-function division and constant handling, were devised, tested and applied successfully in our trials. Simultaneously, a number of general approaches in GP's application, such as strongly typed restraining, fitness calculating and genetic-operation-rate configuring, all of which focus on increasing GP's efficiency, were improved, tested and successfully applied in our trials.

6.2 Future work

In this thesis, the concept of program cloning is proposed not as a solution for a concrete problem, but as a potential approach which might have wide applications in software engineering. This thesis has just stated the reportable work of program cloning; a number of future directions require to be explored.

First, the program-cloning technique itself requires constant improvement. In our trials, only 24 basic programming elements are experimented with for constructing the target programs. Among these 24 programming elements, structure controlling operations include only 4 usages: *If-Else-Then, Sequence, Loop* and *Exception*; data type manipulation considers basic data types, *No-type* and *Array*; and other programming operations include 5 arithmetic operations, 3 logical operations, 5 numerical comparisons 1 assignment operation, and constant and parameters and output facilities. In GP searching process for a target program, many other basic elements for programming such as *switch* and *break* can help construct an ideal program. The programs being cloned in our experiments were all small and simplified in function. Complex and large programs, although they can be simplified by the sub-function dividing method, may expose new challenges in program cloning. Furthermore, when defining the *Array* structure for the *commission* problem, we found that by using objects, data structures can easily be implemented and transformed into GP's primary elements. We believe,

object-oriented programs can be understood and cloned as easy as structural programs.

In the trial version implementation of GP package, limited GP behavioral approaches such as strong typing [Mont95] were adopted and customized for the program-cloning problem; in future work, many other approaches, such as individuals-diversity monitoring, distributed GP searching, individual editing [Koza92] and computational effort statistics, can be tailored and serve the program-cloning problem, which will not only accelerate the program-cloning process but also improve the solution-programs' quality.

Second, practice is needed for applying program cloning to the software engineering domain. As we described in Chapter 5, program cloning can be applied in a series of domains in software engineering as a general approach for program/problem understanding and representing. Through theory discovery based on our trial results, we proposed six potential applications: complexity measurement, software mutant creation, N-version design, test first, test data evaluation and automatic test data generation. Among them, software mutant creation, N-version design, test first and test data evaluation adopt the program-cloning process in a direct and simple way. Hence the research on them, especially on the software mutant creation problem, is relatively easy. Similar work which applying GP in mutation testing has been carried out by Vergilio's group [Emve03]. Adopting program cloning for complexity measurement (see Section 5.1) is well supported by our trials. However, its actualization is a little more complex than that for other domains, and by now, we can only conjecture two measurement guidelines for implement. Adopting program cloning to complexity measurement requires a more mature program-cloning technique. Automatic test data generation, although complex in itself, requires less from the program-cloning technique than complexity measurement. When applying program cloning to automatic test data generation, we need to run the software under test (SUT) as a source of feedback instead of design fitness cases to provide GP evolving environment; and we also assume that distributed computing is definitely helpful in the implementation of this problem.

Finally, as mentioned in Chapter 1, introduction, we have left the problem aside that a cloned program should be expressed in arbitrary languages. A post processor, which translates the solution

programs into any language, is expected to function after the main program-cloning process. In GE [Onei01], because the solution programs are created according to the syntaxes defined by BNF, work on such a post processor is straightforward. In our work on program cloning, individuals were created strictly according to a series of syntaxes as well, yet the syntaxes themselves were not defined and expressed systematically. To compensate for this point, BNF can be borrowed into program cloning's syntax definition. BNF for many popular languages, including C, C++ and Java, are ready and pretty mature, and the work of combining it with program cloning will not only simplify the post processor but also assist in program cloning's syntax definition.

Through this thesis, we have proposed the program-cloning method as a promising program understanding approach. Program cloning's work principles and implementing issues which are based on GP algorithm have been demonstrated. A series of trials which adopt program cloning to automatically create solution programs for their target problems have been described. Furthermore, we discussed program cloning's potential applications and proved that program cloning is a hopeful research direction in software engineering.

Bibliography

[Amkn88] Ammann, P.E. and Knight, J.C., Data diversity: An approach to software fault tolerance, IEEE Transactions on Computers, 37, pp. 418--425, 1988.

[Ange98] Peter J. Angeline. Subtree crossover causes bloat. n John R. Koza et. al., editors, Genetic programming 1998: Proceedings of the Third Annual Conference, pp745-752, Wisconsin, 22-25 July 1998. Morgan Kaufmann.

[Bafn96] W. Banzhaf, F. Francone, and P. Nordin, "The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets," In Proc. 4th Int. Conf. on Parallel Z. Problem Solving from Nature PPSN-96, W. Ebeling, I. Rechenberg, H.-P. Schwefel, H. M. Voigt Z. eds. , Springer: Berlin, 1996, pp. 300-309.

[Banz02] Wolfgang Banzhaf, William B. Langdon: Some Considerations on the Reason for Bloat. Genetic Programming and Evolvable Machines 3(1): 81-91 (2002).

[Beck02] K. Beck, Test Driven Development: By Example, Addison Wesley, 2002.

[Beck99] K. Beck, Extreme Programming Explained: Embrace Change, Addison Wesley, 1999.

[Blbr01] S. Bleuler, M. Brack, L. Thiele, E. Zitzler. Multiobjective Genetic Programming: Reducing Bloat Using SPEA2. Congress on Evolutionary Computation (CEC-2001), pp 536-543, May 2001.

[Brlw02] L. Briand, Y. Labiche, Y. Wang, Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts, Technical Report SCE-02-09, Carleton University, October 2002.

[Broo86] Frederick P. Brooks, Jr. No Silver Bullet Essence and Accidents of Software Engineering, Information Processing 1986, ISBN No. 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V. (North-holland) IFIP 1986.

[Buht00] Bull, L., Hurst, J. & Tomlinson, A. (2000) Self-Adaptive Mutation in Classifier System

Controllers. In J-A. Meyer, A. Berthoz, D.Floreano, H. Roitblatt & S.W. Wilson (eds) From Animals to Animats 6 - The Sixth International Conference on the Simulation of Adaptive Behaviour, MIT Press, pp 460-467, 2000

[Chtz02] T.Y. Chen, T.H. Tse, Z. Zhou. Semi-Proving: an integrated method based on global symbolic evaluation and metamorphic testing. ACM Press, New York, pp. 191-195, 2002.

[Coll03] Terry Colligan, There is No Silver Bullet for C Programming Problems, http://www.tenberry.com/ic/nosilver.htm, October 2003.

[Corp96] Parasoft Corporation, Mutation Testing: A New Approach to Automatic Error-Detection. http://www.parasoft.com/jsp/products/article.jsp?articleId=291, 1996

[Dama88] Dalal, S.R. and Mallows, C.L., When should one stop testing software, Journal of American Statistical Association, 83:872-879, 1988.

[Dels79] R. Demillo, J Lipton, F Sayward, Program mutation: a new approach to program testing, Infotech State of the Art Report, Software Testing, Volume 2, pp. 107 – 126, 1979.

[Demi78] Demillo, R.A., Lipton, R.J. and Sayward, F.G., Hints on test data selection: Help for the practicing programmer, Computer 11, 4, 34-43, 1978.

[Emve03] M.C. Emer and S.R. Vergilio, Selection and Evalutation of Test Data Based on Genetic Programming, Software Quality Journal, 11, 167-186, 2003.

[Fowl00] M Fowler, The New Methodology, http://www.martinfowler.com/articles/newMethodology.html, 2000.

[Gell95] Gell-Mann, M, What is complexity, Complexity, Vol. 1, No. 1, pp 16 -19, 1995.

[Gibb94] W. W. Gibbs. Software's chronic crisis, Scientific American (International Edition), pages 72--81, Sept. 1994.

[Gito02] Giacobini M. and Tomassini M. 'Limiting the Number of Fitness Cases Using Statistics', in 'Proceedings of the 2002 Genetic and Evolutionary Computation Conference Workshop Program', Alwyn Barry Ed., New York City (New York, USA), 2002.

[Gumn96] Carl Gunter, John Mitchell and David Notkin, Strategic Directions in Software Engineering and Programming Languages, ACM Computing Surveys, Vol. 28, No. 4, December 1996.

[Hahd00] M. Harman, R. M. Hierons and S. Danicic, 2000, the relationship between program dependence and mutation testing, Mutation 2000, pp. 15-23, 2000.

[Hals77] Halstead, M.H., Elements of Software Science, New York, Elsevier North-Holland, 1977.

[Hihd99] R.M. Hierons, M. Harman and S. Danicic, Using Program Slicing to Assist in the Detection of Equivalent Mutants, The Journal of Software Testing, Verification, and Reliability, 9(4), pp. 233-262, 1999.

[Jorg02] Paul C. Jorgensen, Software testing – A craftsman's approach, 2nd edition, Chapter 2, Examples: p 17 – p 26, CRC Press LLC, 2002.

[Knle86] Knight, J.C., Leveson, N.G., An experimental evaluation of the assumption of independence in multiversion programming, IEEE Transaction on Software Engineering, Vol. 12, No. 1, pp. 96 – 019, 1986.

[Koza92] J. Koza. Genetic Programming. MIT Press, Cambridge, MA, 1992.

[Kush02] Ibrahim Kush, Genetic Programming and Evolutionary Generalization, IEEE Trans. Evolutionary Computation, Vol. 6, No. 5, October 2002.

[Lapo97] W. B. Langdon and R. Poli, "Fitness causes bloat," in Soft Computing in Engineering Design and Manufacturing. P. K. Chawdhry, R. Roy, and R. K. Pant (eds.), Springer-Verlag: London, 1997, pp. 13–22.

[Luke02] Sean Luke, http://www.cs.umd.edu/projects/plus/ec/ecj, Jan 2002.

[Lupa02] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure, Parallel Problem Solving from Nature, International Conference 7th: pp 411- 421, Jan 2002.

[Lyhe93] Lyu, M. R. and He, Y., Improving the N-Version Programming Process Through the Evolution of a Design Paradigm, IEEE Transactions on Reliability, Vol. R-42, 1993, pp. 179-189.

[Mcca76] McCabe, T., A Complexity Measure, IEEE Transactions on Software Engineering, SE-2, No. 4 (December): 308-20, 1976.

[Mill72] Mills, H.D., On satatistical validation of computer programs, IBM Rep. FSC72-6015, Federal Systems Devision, IBM, Gaithersburg, Md, 1972

[Mims01] Christoph C. Michael, Gary McGraw and Michael A. Schatz, Generating software test

data by evolution, IEEE transactions on software engineering, vol. 27, NO. 12, December 2001.

[Mirc04] James Miller, Marek Reformat, Xinwei Chai, On the possibilities of (pseudo-) software cloning from executable specifications or source code, Empirical Software Engineering, Under Review

[Mont95] D.J. Montana. Strongly typed genetic programming. Evolutionary Computation, 3(2): 199-230, 1995.

[Mrbo99] Mresa E. S. and Bottaci L., Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study, The Journal of Software Testing, Verification, and Reliability, 9(4), pp. 205-232, 1999.

[Oflr96] Offut A. J., Lee A., Rothermel G., Untch R. H. and Zapf C., An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering and Methodology, 5(2), pp.99-118, 1996.

[Ofpa97] J. Offutt, J. Pan, Automatically Detecting Equivalent Mutants and Infeasible Paths, The Journal of Software Testing, Verification, and Reliability, 7(3), pp. 165 – 192, 1997.

[Onei01] O'Neill M. Automatic Programming in an Arbitrary language: Evolving Programs with Grammatical Evolution. Ph. D. thesis, University of Limerick, 2001.

[Onry01] O'Neill M., Ryan C. Grammatical Evolution. IEEE Trans. Evolutionary Computation, Vol. 5 No. 4, August 2001.

[Ould91] M. Ould, "Testing---a challenge to method and tool developers," Software Engineering Journal, vol. 6, pp. 59--64, Mar. 1991.

[Pine88] Pines, D., Emerging synthesis in science, Addison-Wesley, 1988.

[Rawe85] Rapps. S. and Weyuker, E.J., Selecting software test data using data flow information, IEEE Transactions on Software Engineering SE-11(4): 367-375, 1985.

[Recm03] Marek Reformat, Xinwei Chai and James Miller, Experiments in automatic programming for general purposes, The15th IEEE International Conference on Tools with AI, Nov. 2003

[Sofd96] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pages 215-223, Stanford University, CA, USA, 28-31 July 1996. MIT Press.

[Soul98] T. Soule. Code growth in genetic programming. PhD thesis, University of Idaho. 1998.

[Spin01] Spinoza, E. et al. Chameleon: A generic tool for genetic programming, In Proceedings of the Brazilian Computer Society Conference, Forteleza, Brazil, August 2001.

[Tean97] Astro Teller, David Andre, Automatically Choosing the Number of Fitness Cases: The Rational Allocation of Trials, Genetic Programming 1997: Proceedings of the Second Annual Conference, pp 321--328, 1997.

[Torr00] Torres-Pommales, W., Software Fault Tolerance: A Tutorial, NASA/TM-2000-210616, 2000.

[Trac00] Nigel James Tracey, A search-based automated test-data generation framework for safety-critical software. September 2000.

[Vard91] Vardi, I. "The 3x+1 Problem." Ch. 7 in Computational Recreations in Mathematica. Redwood City, CA: Addison-Wesley, pp. 129-137, 1991.

[Watk95] Alison Lachut Watkins., The automatic generation of test data using genetic algorithms., Proceedings of the 4th Software Quality Conference, 2:300-309, 1995.

CURRICULUM VITAE

Xinwei Chai
E-mail: chaixv@ece.ualberta.ca
W5-040 9107-116 Street Edmonton, Alberta, Canada T6G 2V4

Education

Jan. 2002 – Dec. 2003 **MSC Electrical & Computer Engineering**
University of Alberta, Canada

Aug. 1991 – Jul. 1995 **BENG Computer Science and Technology**
Xidian University, China

Professional experience

Nov. 1996– Jul.2001 **Software Developer**

Designed and developed E-Commerce-solution products focusing on middle tier components and security mechanism, developed embedded software for consumptive electronic products

Jul. 1995 – Oct. 1996 **Assistant Project Engineer**

Designed and configurated distributed control systems based on mature DCS products, supervised projects and provided clients maintains

Research:

The research interests lay in the area of software Engineering such as Automatic Software Testing, Automatic Programming, Software Quality Assurance, Software Process and Modeling and Software Security, as well as Evolutionary Algorithms such as Genetic Algorithm, Genetic Programming and Simulated Annealing

Publications:

- Marek Reformat, Xinwei Chai and James Miller, Experiments in automatic programming for general purposes, The15th IEEE International Conference on Tools with AI, Nov. 2003
- James Miller, Marek Reformat, Xinwei Chai, On the possibilities of (pseudo-) software cloning from executable specifications or source code, Empirical Software Engineering, Under Review